Editors: Paul F. Dubois, paul@pfdubois.com
David Beazley, beazley@cs.uchicago.edu

WHY JOHNNY CAN'T BUILD

By Paul F. Dubois, with Thomas Epperly and Gary Kumfert

IN MY 10 YEARS OF EDITING THE SCIENTIFIC PROGRAMMING DEPARTMENT, I'VE TRIED TO SELECT ARTICLES THAT GIVE YOU REAL INFORMATION YOU CAN USE OR THAT SHOWCASE NEW techniques and ideas. This installment is a departure; it discusses an essential problem I don't know how to solve.

The title of this article refers to Rudolph Flesch's famous 1955 book, *Why Johnny Can't Read*, which called attention to a nationwide decline in reading ability. Here, I want to talk about another situation in which an important ability is lacking: the ability to create significant, portable scientific software. I'll discuss some of the reasons this problem exists and suggest some approaches to solving it that seem promising.

The Birth of the Build Problem

The "build problem" is the task of creating a system that delivers a software system on a wide variety of platforms that have various programming tools and configurations. Unfortunately, the build problem's dimensions are hard to measure. Thomas Epperly and Gary Kumfert did a survey of code groups working in a variety of disciplines at Lawrence Livermore National Laboratory.[1] Their results suggested that of the effort spent on modern scientific code, perhaps 10 and up to 30 percent of it (for small and large projects, respectively) went into trying to solve the build problem. My own experience confirms this estimate. If these estimates are anywhere near accurate, the build problem represents a substantial part of scientific simulation budgets, yet almost nothing is being done about it.

The build problem's significance has grown in part from successes in other areas of the program construction process. The object-oriented revolution and the earlier modular programming movement led to the idea of programs constructed from existing reliable pieces, often supplied by outside "vendors." Moreover, the open-source movement and the Internet's continued development have made obtaining such pieces and contributing to their evolution possible. Whereas a major scientific program might have been nearly self-contained a decade ago (with references to only a few mathematical or graphical libraries), a modern program could rely on 20 or more outside packages and program-building tools.

A code I work on, for example, uses 22 tools and libraries that the code team did not write (not counting those things usually available with the system such as compilers, linkers, shells, and so on). It includes

- six mathematical packages,
- two packages for accessing physical property databases,
- two physics algorithm packages,
- one graphics library, and
- eleven computer-science tools and libraries.

Ultimately, this list means that we must build 22 tarballs on whatever variety of platforms we have (in our case, everything from Linux to massively parallel processors) using several different compilers, linkers, and other tools and system libraries.

The problem we run into is that few of the people supplying outside software have access to the same set of platforms and tools we're using. Even when all else is equal, different system administrators configure file systems differently, supply different default startup files and environment variables, and keep various parts of the systems up to date at different rates. The system manufacturers don't have identical header files, file locations, or tool interfaces—rather, they have tools with different limitations and quirks. Even the commands to unpack the tarball can vary.

This situation is rather frustrating: just when we're starting to solve the problem of how to create software using reusable parts, it founders on the nuts-and-bolts problems outside the software itself.

Three Parts of the Process

The build process essentially consists of three steps: configuration, compilation, and linking. In theory, they should follow each other seamlessly. In practice, though, the line between compilation and linking blurs for C++ codes, and other linking failures can occur at runtime when shared libraries are

Dave's Sideshow

THE WAR ON TERROR—UH, I MEAN SPAM

Lately, it seems like I've done very little real work other than delete spam from my mailbox, delete spam from email lists, curse about spam under my breath, and generally whine about spam to anyone who would care to listen (which hasn't been that many people, I'm sorry to report). I received a modest 11,000 spam email messages last year, and this flood shows no signs of abating. Sigh.

For a while, I considered the idea of adopting a write-only approach to email, but that really didn't seem too practical (although the idea of sending out an occasional random no-reply email proclamation to people seemed curiously attractive). In the meantime, I continue to suffer from spam overload.

The current spam situation makes me long for the simple days when I used to be the sysop of a dial-up BBS system about 20 years ago. Life was good as a sysop. People from all over would dial up and post interesting messages. Sometimes they would ring to chat. If a user were annoying, you could just delete his account. And you could have loads of fun playing mind games with would-be hackers and software pirates.

Probably the high point of my sysop career was the secret modification I made to my BBS software that let me tag certain users as pests (the infamous "pest flag"). The pest flag was reserved for that especially vile class of users—you know, the ones who always shouted incoherent messages in all-caps or constantly posted insulting drivel for no apparent reason other than inciting flame wars. The pest flag was also unique in that rather than sim-

used. Complicating matters further, the build process also must be compatible with the source-code control process.

Configuration is the mechanism for handling differences in computing environments (that is, the set of all things that can differ from one target location to another). Perhaps the most-used tool in this area is `autoconf`, a part of the GNU project. The information the configuration tool collects goes to a build tool, which must recompile and relink those parts of the software that must be rebuilt.

Because users build their software repeatedly after they make small changes, calculating dependencies and subsequently rebuilding as little as necessary is a must. The standard tool used here is `make`, one of the oldest parts of our infrastructure. Actually, there is no single `make`—rather, there's a large set of programs with the same name and general behavior written by various computer vendors, along with a GNU variant called `gmake`. On some platforms, users employ integrated environments instead. Getting software into or out of Microsoft Windows, for example, is notoriously difficult. Likewise, various Macintosh platforms require different treatment.

Unfortunately, neither `make` nor `autoconf` is easy to use. Large numbers of scientists do not use `autoconf` or understand it; it is strictly a tool for geeks. The `make` program is notoriously poor at dealing with different environments and situations such as one process producing two products; it has spawned many attempts to enhance or replace it, such as `Imake` and `Jam` (see the related sidebar).

Who Can We Blame?

Is this mess simply inevitable? Not entirely—it's artificially difficult for several reasons.

First, language standards committees deliberately do not deal with compilers, just the languages they must compile. They don't know or care if a compiler has a debug mode, or how you invoke it, or whether invoking it and some other option will be illegal. Likewise, such committees specify almost nothing about what a compiler will do to implement the language. For example, the Fortran 90 committee did not say anything about the data structures needed to pass array descriptors or the mechanism by which the module interface information would be retained. As a result, writing portable interfaces to C routines or creating a portable input file to `make` is difficult.

The intellectual justification for this position is that to specify such matters would reduce potential compiler authors' creativity. Is this benefit worth the harm it causes? Another factor is that committees have significant representation from compiler vendors. Agreeing that things should be done one way causes expense and difficulty for those who currently do it another way. In short, we users are not part of the process—just victims of it.

Second, linker technology is not standardized, and the tools connected with it are horrible. Add the issues involved with shared libraries, and you have a total mess. Let's say that you finish building a program and it tells you something about a ply deleting a user's account or restricting permissions, it just made everything the pest performed completely invisible to everyone else. Post an idiotic message, and nobody else ever sees it. Send email, and it's silently never delivered. Upload a file, and it never actually gets stored.

Of course, the key to implementing a really good pest flag is ensuring that the pest remains unaware of his or her special status. When pests log on to the system, their messages should still appear as normal, sent email should say that it was received, and uploaded files should still appear to be there. Invariably, pests would become rather bored with the community and leave—mission accomplished. If you ask me, the Internet could use a pest flag right now.

Reproducibility of Results

Reproducing the source code used to produce a scientific result is a subtle, and usually overlooked, aspect of scientific data management. Oddly enough, this problem usually only arises at the most inconvenient times: when you're writing a paper and you're trying to summarize some work you did six months earlier, for example. The only trouble is remembering all the details of what you were doing at the time—especially if you've made lots of code modifications. If you've encountered similar problems and implemented a solution, I'd definitely like to know about it. Better yet, consider writing an article!

The Aesthetics of Programming

In my spare time, I recently decided to take some music classes in an attempt to learn how to play jazz piano. What does this have to do with scientific programming, you ask? Well, nothing, and I guess that's the whole point. In any case, as part of the class we are often presented with certain artistic challenges—for example, to create an engaging improvised solo using nothing but two notes or perhaps nothing but long notes. Needless to say, this is a lot harder than it sounds, and the end result rarely sounds as good as you would like (at least it doesn't for a novice like me).

I'm starting to wonder whether this kind of approach would work for an introductory programming class. I could force students to write programs with all sorts of weird constraints like, "you're not allowed to use the multiplication operator" or "the `goto` statement is the only form of control flow—oh, and by the way, you're only allowed to use one function too." Needless to say, this would probably force students to come up with devious solutions. Imagine the chaos and puzzled looks. Hmmm. Stay tuned.

routine being missing—only you've never heard of this routine, and its name looks like it's written in Martian. What are you to do? Even worse, what if it tells you this only at runtime?

Third, as a consequence of Microsoft's monopolistic hold over business software, many business programmers do not face so many portability issues, which means market share is inadequate to induce progress in this area. On the contrary, most vendors have attempted to "hook" users with their own language extensions, tool sets, and development environments. Naive users often think that the language standard is what their compiler will compile; they realize only later that their product is painfully nonportable.

Even hooks have their problems: inadequate standards for them exist between tools. It is quite common to want (or even need) to specify linker options on the compile line, for example, but there's no standard way to do so. More seriously, tools that check for memory use errors often want to intervene in the link process, but there is no standard way to notify the linker of that.

Finally, computer scientists on certain platforms can use tools such as `autoconf` or `Imake`, albeit with some difficulty. They frequently say to me that `autoconf` is "good enough," without seeming to realize how totally geeky it is. Normal people—I'm talking people with PhDs in engineering, physics, and mathematics—can't use these things. After all, there wouldn't be much of an auto industry if you had to be a mechanic to own a car.

Smaller Problem, Better Tools

Solving the build problem will require a two-pronged approach: making the problem smaller and improving the tools that solve the problem. To make the problem smaller, we could

- get standards expanded to cover what needs to be covered (we'd have to account for fewer differences);
- improve testing and debugging methodologies (tools would have fewer idiosyncratic bugs to work around);
- use the Internet in more creative ways (we could acquire knowledge about how to build packages on certain platform–compiler combinations rather than having to use tools such as `autoconf`); and
- increase our use of component approaches such as Babel (we could avoid language interface problems).[2,3]

To improve our tools, let's start with the linker. As far back as 1976, Lawrence Livermore National Laboratory had a locally written link tool that told you exactly what was missing, who asked for it, and exactly what routines loaded from where. It operated on a routine level (not an object-file one), and it didn't complain about something being missing if a library asked for a routine in a library listed before it on the link line. It wasn't perfect, but it was miles ahead of the standard Unix link. A step forward would be to build a better, standardized linker.

Such a linker could be key to solving another problem: de-

Open-Source Software for Building and Configuration

Autoconf (www.gnu.org/manual/autoconf/) is the most widely used configuration tool, but it's hard for authors to learn and often confuses users.

`make` is a build tool that comes with every version of Unix, but every version differs in some way. The Windows version is called `nmake`, but most Windows users use the Visual C++ environment with the build tool integrated into it.

`Imake` (www.dubois.ws/software/imake-stuff/) was something my group tried for a year or so, but again it was too hard for scientists. (This Web site by Paul Dubois is not my Web site. As Yoda said, there is *another*.)

Cons (www.dsmit.com/cons/) is a Perl-based system about which some people think highly.

SCons (www.scons.org) is discussed more in the main article. It is a descendant of Cons. Some configuration is now supported too.

`JAM/MR` (perforce.com) is another build system that has attracted an enthusiastic audience.

Xenofarm (www.lysator.liu.se/xenofarm/) is a new attempt to centralize the build knowledge using a client-server architecture. It was derived from a method called Tinderbox used by the Mozilla team.

bugging. Writing good, portable debugging tools, especially those that deal with multiple languages, is hard. For example, David Beazley's work on a Python–C integrated debugger required unreasonable effort.[4]

Some open-source programs are close to universally portable thanks to the efforts of a large, diverse user base. The growth of programming languages such as Python and Perl come to mind here. These packages in turn, use configuration to run the C compiler and linker, which are the most portable of underpinnings. It would seem in everyone's best interests if we could base our tool set on just this set of portable tools.

Los Alamos National Laboratory sponsored a "Software Carpentry" design contest in January 2000 to elicit new ideas and tools in a variety of areas related to software construction. The Software Carpentry contest was based on the idea of making Python-based tools, because Python is easy to read and learn. If tools are written in an accessible language such as Python, the set of contributors to them opens up to include more of us. The winning designs in every category emphasized user customization and extendibility. Two of the contest's categories were a platform-inspection tool to replace `autoconf` and a build-management tool to replace `make`.

The fact that these were separate categories was controversial—the boundary between the tools is part of the problem rather than part of the solution. SCons (http://scons.sf.net) won the `make` tool contest. SCons does part of the configuration job and is increasingly doing more. The basic ideas behind it come from Cons, a Perl-based tool. Perl, unfortunately, is a difficult language for most scientists. After the design contest, Steven Knight and others implemented the SCons design and have since attracted a growing audience of users.

SCons: A Modern Build Tool

I installed SCons on both a Linux box and a Windows XP box by unpacking the tarball and executing

```
python setup.py install.
```

After installing SCons on Windows, you need to make a `scons.bat` script to execute the `C:\Python23\Scripts\scons` script, as is usual with Python applications, and put it in your path. (If you don't have Python you can get it from www.python.org. For Windows, there is the usual prebuilt installer.)

SCons admits from the first that building programs is always going to involve special cases: a certain file that cannot be compiled with optimization when using a certain compiler, a local preprocessor that must be invoked, a tool that produces multiple products, or a new language that SCons' authors have never heard of.

Although it's designed to be extendible and customizable, SCons keeps simple things fairly simple. It is written entirely in Python, which is pretty much everywhere. SCons also tries to make it possible to write one input file to guide the build on a variety of platforms, even though these platforms might have different naming conventions for things such as object files or library names.

Just as `make` looks for its input in a file named `Makefile`, SCons operates off a user-written file that explains what is to be built. This file is typically named `SConstruct`. Assuming we have written a "Hello, world" program in file `hello.c`, the `SConstruct` file needed to build it would look like this:

```
env = Environment()
env.Program('helloworld', ["hello.c"])
```

The first line constructs an environment, which you can customize to build files a certain way. In this case, we add a target to the default environment, which is an executable program called `helloworld` whose sources are given in a list supplied as the second argument.

If we enter the command `scons`, the executable will be built. Here it is on Linux:

```
> scons
```

```
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -c -o hello.o hello.c
c++ -o helloworld hello.o
scons: done building targets.
```

`SConscript files` refers to the `SConstruct` file and any files included in it via `SConscript` commands. Also, SCons can traverse subdirectories or restrict itself to specific targets.

With `make`, moving to Windows would be a job, but with SCons it just works:

```
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cl /nologo /c hello.c /Fohello.obj
hello.c
link /nologo /OUT:helloworld.exe hello.obj
```

Note that on Windows, SCons appropriately renamed my target to `helloworld.exe`, knew that Windows object files are named ".obj" not ".o", and knew the right way to run the Visual C++ compiler and linker via the command line. SCons's design contains features like this to reduce the differences between systems. We'll see another example shortly.

If I add the argument `-debug=tree` to my SCons command, I get a listing of my dependency tree:

```
+-.
 +-SConstruct
 +-hello.c
 +-hello.obj
 | +-hello.c
 +-helloworld.exe
 | +-hello.obj
 | +-hello.c
 +-stdio.h
```

And I don't even need a "clean" target because: `scons -c` does the job:

```
> scons -c
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Cleaning targets ...
Removed hello.o
Removed helloworld
scons: done cleaning targets.
```

Suppose now that I want to put the calculation of the message into a static library, with header file `hello.h` and source file `hello.c` in a subdirectory `src`. I modify the original `hello.c` in this directory to include `hello.h` and call the function that it defines to get the message. I modify `SConstruct` like so:

```
env = Environment(CPPPATH=['src'],
       LIBS='hlib', LIBPATH='.')
env.StaticLibrary('hlib', ['src/hello.c'])
env.Program('helloworld', ['hello.c'])
```

SCons scans my input files for dependencies. I tell it where to look for the include files using the `CPPPATH` variable of the SCons environment `env`. Now when I run `scons` I get

```
> scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
cc -Isrc -c -o hello.o hello.c
cc -Isrc -c -o src/hello.o src/hello.c
ar r libhlib.a src/hello.o
ranlib libhlib.a
c++ -o helloworld hello.o -L. -lhlib
scons: done building targets.
```

Again, this same `SConstruct` file will work on Windows. Of course, moving the specification for the library into a subsidiary `SConscript` file in the `src` directory is straightforward, and when working in the `src` subdirectory, you can rebuild just that library or the whole program. Note that nowhere did I have to know that the library I wanted to call `hlib` would be `libhlib.a` on this platform. That's good, because on other systems, this wouldn't be the library's name.

SCons also supports parallel builds with the `-j` flag, similar to `gmake`.

Already SCons supports quite a few languages and other tools, notably including Latex and PDF file creation, f77, and Java. It has extensive facilities for allowing options and choices of tools, and by using it, you can set up many different targets with different environments. Most importantly, it is designed to let you add your own build tools and dependency scanners, so that the system is open rather than closed.

One hint when looking at the SCons source and documentation: the examples are good, but hard to find. You'll find them hiding near the end of the main page for `scons`. One thing to keep in mind is that the `SConstruct` file is a Python input file, so you have the full power of a programming language—you can decide to include a certain file or not on a given platform, or you can decide to use a certain compiler flag.

Despite the superiority of Flesch's phonics method, he later had to write a book entitled, *Why Johnny Still Can't Read*. New tools, even if far superior, cannot and will not be adopted very quickly. Using the dual approach of reducing the problem while improving our tool set is important and offers a greater chance at success.

In 1984, I began writing Basis (http://basis.llnl.gov), which was probably the first widely used steering system for scientific programs. That year, I read a business-school paper that claimed the average time for a new technology's widespread adoption was 17 years. I laughed. Surely, I thought, this modern generation of ours wouldn't take that long. We were about to have the Internet, the world would live on Internet time, and the government would give me stock options.

I stopped laughing in 2001, when it dawned on me that steering was just then becoming widespread in scientific programming.

Although it might take some time, we have to start somewhere. Otherwise, we'll look back in 20 years and discover that Johnny and Jane still can't build. We can do a lot as a community, and government and the industry can do much to help. Government and industry have done a lot to encourage the standards process and the creation of pioneering technology; they just have to push harder to remove artificial barriers to progress.

References

1. G. Kumfert and T. Epperly, *Software in the DOE: The Hidden Overhead of "The Build,"* UCRL-ID-147343, Lawrence Livermore Nat'l Lab., 2002.
2. S. Kohn et al., "Divorcing Language Dependencies from a Scientific Software Library," *10th SIAM Conf. Parallel Processing*, SIAM Press, 2001, CD-ROM.
3. T. Epperly, S. Kohn, and G. Kumfert, "Component Technology for High-Performance Scientific Simulation Software," *Working Conf. Software Architectures for Scientific Computing Applications*, 2000, Int'l Federation for Information Processing, CD-ROM.
4. D. Beazley, "WAD: A Module for Converting Fatal Extension Errors into Python Exceptions," *Proc. 9th Int'l Python Conf.*, Foretec Seminars, 2001, pp. 1–10.

Paul Dubois is a computer scientist in Lawrence Livermore National Laboratory's Center for Applied Scientific Computing. His career includes pioneering work on computational steering and the use of object technology for scientific programming. He is a coeditor of *CiSE*'s Scientific Programming department. He has a PhD in mathematics from the University of California, Davis.

Thomas Epperly is a computer scientist in Lawrence Livermore National Laboratory's Center for Applied Scientific Computing. His research interests focus on software frameworks and standards for large-scale computational modeling of physical systems. He has a BS chemical engineering from Carnegie-Mellon University and a PhD in chemical engineering from the University of Wisconsin. Contact him at tepperly@llnl.gov.

Gary Kumfert is a computer scientist in Lawrence Livermore National Laboratory's Center for Applied Scientific Computing. His research interests include component technology for scientific programming and model coupling on massively parallel computations. He has a BS in applied mathematics and a PhD in computer science, both from Old Dominion University. Contact him at kumfert1@llnl.gov.

Correction

Last issue's Scientific Programming ran the incorrect email address for John Reid. His correct email address is j.k.reid@rl.ac.uk. We regret any confusion this error might have caused.

Submissions: Send two copies, one word-processed file and one PostScript file, of articles and proposals to Francis Sullivan, Editor in Chief, *CiSE*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314; cise@computer.org. Submissions should not exceed 6,000 words and 15 references. All submissions are subject to editing for clarity, style, and space.

Editorial: Unless otherwise stated, bylined articles and departments, as well as product and service descriptions, reflect the author's or firm's opinion. Inclusion in *CiSE* does not necessarily constitute endorsement by the IEEE, the AIP, or the IEEE Computer Society.

Circulation: *Computing in Science & Engineering* (ISSN 1521-9615) is published bimonthly by the AIP and the IEEE Computer Society. IEEE Headquarters, Three Park Ave., 17th Floor, New York, NY 10016-5997; IEEE Computer Society Publications Office, 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1314, phone +1 714 821 8380; IEEE Computer Society Headquarters, 1730 Massachusetts Ave. NW, Washington, DC 20036-1903; AIP Circulation and Fulfillment Department, 1NO1, 2 Huntington Quadrangle, Melville, NY 11747-4502. Annual subscription rates for 2003: $42 for Computer Society members (print only) and $48 for AIP member society members (print plus online). For more information on other subscription prices, see http://computer.org/subscribe or www.ezsub.net/cise. Back issues cost $10 for members, $20 for nonmembers. This magazine is available on microfiche.

Postmaster: Send undelivered copies and address changes to Circulation Dept., *Computing in Science & Engineering*, PO Box 3014, Los Alamitos, CA 90720-1314. Periodicals postage paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 0605298. Printed in the USA.