Editors: Paul F. Dubois, paul@pfdubois.com
George K. Thiruvathukal, gkt@cs.luc.edu

BUILDING SOFTWARE WITH SCONS

By Steven Knight

SOFTWARE CREATION IS A COMPLICATED PROCEDURE. THE PROLIFERATION OF COMMERCIAL AND OPEN-SOURCE PACKAGES MEANS THAT A TYPICAL SOFTWARE PACKAGE might have to know how to link to dozens of different libraries or other third-party software. It might even have to build some of those libraries from scratch, not to mention build itself (and the libraries) in many different versions for various operating systems and hardware platforms.

A next-generation software build tool called SCons can greatly simplify the headaches involved in building complicated software projects. I'll demonstrate it by building a sample project that involves source code spread across multiple directories. We'll also build two versions of an external software package to link against: a debug version and an optimized version.

SCons Basics

Before we get to the sample project, let's explore SCons. (More detailed information about it, including downloads, installation instructions, and documentation, is available at www.scons.org.)

The most distinctive thing about SCons is that its configuration files are actually Python scripts; to specify dependencies, we call various SCons functions using normal Python syntax, rather than a special-purpose language. The top-level configuration file—the SCons equivalent of the `Makefile` at the top of a source tree—is called `SConstruct`.

Consider the following example, which builds a program called `myprog`:

```
Program(target = 'myprog',
        source = ['file1.c', 'file2.f',
                  'file3.y'],
        CC = 'gcc',
        CCFLAGS = ['-g'],
        CPPDEFINES = ['DEBUG'],
        CPPPATH = ['include'],
        FORTRAN = ['f90'],
        FORTRANFLAGS = ['-X'],
        LIBS = ['m', 'foo'],
        LIBPATH = ['libs'])
```

This example demonstrates many of the SCons features we'll use in the more complicated example. Specifically, line by line:

- We leave off any program suffix on the target file name. When compiling on Windows, SCons will automatically append the `.exe` suffix for us and build `myprog.exe`.
- The source file list can contain a mixture of languages—in this case C, Fortran, and Yacc. By default, SCons will invoke the right compilers with the right options to build each type of source file.
- The `CC` variable tells SCons what C compiler to use. If we had left off this definition, SCons would have tried to pick a reasonable default compiler based on what's installed on our system.
- The `CCFLAGS` variable contains specific compilation options—in this case, the `-g` option that includes debug information.
- The `CPPDEFINES` variable tells SCons what variables should be defined on the command line. Note that we don't do this by explicitly specifying `-D` options by hand in `CCFLAGS`. This allows SCons to construct the right command line for different types of compilers—for example, `-DDEBUG` on Linux systems, `/DDEBUG` on Windows.
- The `CPPPATH` definition tells SCons where to look for header (`.h`) files. Notice that we don't specify `-I` options on the command line; SCons will create them for us.
- The `FORTRAN` variable tells SCons which Fortran compiler to use. Again, if we had left this blank, SCons would have tried to pick a reasonable default based on what's installed on the system. SCons has very flexible Fortran support, including the ability to independently configure compilers and options for Fortran 77, 90, and 95.
- The `FORTRANFLAGS` variable specifies Fortran compilation options.

```
src/
├──SConstruct
├──libx/
│  ├──SConscript
│  ├──file1.c
│  ├──file2.c
│  └──file3.c
└──progy/
   ├──SConscript
   ├──file4.c
   ├──file5.c
   └──file6.c
```

Figure 1. Source tree. In this initial layout of a multidirectory SCons example, the SCons configuration files are in bold.

- The `LIBS` variable is a list of libraries to be linked with this program. Note that we specify only the base name of the library, without a `"lib"` prefix or `".a"` suffix and without a `"-l"` flag. This allows SCons to add the right command-line options to link with the specified libraries on any operating system.
- The `LIBPATH` variable tells SCons where to find the libraries. Again, we don't specify a `-L` prefix, but let SCons add the appropriate option for our operating system and linker.

Notice that we specified all these variables as Python keyword arguments to the `Program()` function. (In SCons terms, the `Program()` function is called a *builder*.) If we wanted to build multiple programs, it would be tedious to repeat all those variables every time we call `Program()`. SCons lets us collect them into a *construction environment* through which we can then call `Program()` multiple times to build multiple programs with the same compilers and options:

```
env = Environment(CC = 'gcc',
                  CCFLAGS = ['-g'],
                  CPPDEFINES = ['DEBUG'],
                  CPPPATH = ['include'],
                  FORTRAN = ['f90'],
                  FORTRANFLAGS = ['-X'],
                  LIBS = ['m', 'foo'],
                  LIBPATH = ['libs'])
env.Program('myprog', ['file1.c', 'file2.f',
                       'file3.y'])
env.Program('prog2', ['file4.c', 'file5.c',
                      'file6.c'])
```

We can create multiple construction environments, make copies of them, and tailor them for different purposes.

Building in Multiple Directories

I'll begin explaining our more complicated example by demonstrating how to build software that's spread across multiple directories.

Within any SCons configuration file, we call the `SConscript()` function to specify one or more subsidiary configuration files that SCons should read. By convention, these files are named `SConscript` (hence the name of the function), although we can name them anything we like. All SCons configuration files are generically referred to as *SConscript files*, regardless of the actual file name.

The typical configuration is to place one subsidiary SConscript file in each subdirectory, allowing us to keep each build configuration file next to the source-code files it's supposed to build. This is also the way large projects typically set up their `Makefiles`, but unlike Make, SCons doesn't reinvoke itself recursively in each subdirectory. Instead, it reads up all the configuration files into one global view of the dependencies between the various target and source files, and then builds the requested targets—along with their dependencies—as efficiently as possible.

Another way in which SCons differs from Make is that the variables defined in each SConscript file aren't automatically shared between files. This is to avoid the Make phenomenon of changes in one configuration file having unintended side effects in other configuration files. Instead, SCons requires us to explicitly `Export()` variables from configuration files so that other configuration files can `Import()` them.

We'll build an example that consists of source code in two subdirectories. The `libx` subdirectory contains the source of an internal library that we want to build and link with, and the `progy` subdirectory contains the source of the program we're building. To keep things simple, each subdirectory contains three numbered C source files, although recall from earlier that we can build multiple languages just by listing the source files. We'll have one top-level `SConstruct` file and one `SConscript` file in each of the two subdirectories, so the initial layout of our source tree looks like Figure 1.

The top-level SConstruct file for Figure 1 might look like this:

```
# Top-level SConstruct file that calls
# subsidiary SConscript files to build
# a library and a program.
env = Environment(CC = 'gcc',
                  FORTRAN = 'g77')
Export('env')
SConscript('libx/SConscript')
SConscript('progy/SConscript')
```

This SConstruct file sets up a base construction environment containing the compilers we want to use for all compilations throughout the source tree. Here we've set both C

```
src/
 ├──SConstruct
 ├──libx/
 │   ├──SConscript
 │   ├──file1.c
 │   ├──file2.c
 │   ├──file3.c
 │   ├──file1.o
 │   ├──file2.o
 │   ├──file3.o
 │   └──libx.a
 └──progy/
     ├──SConscript
     ├──file4.c
     ├──file5.c
     ├──file6.c
     ├──file4.c
     ├──file5.c
     ├──file6.c
     └──progy
```

Figure 2. Building an example. The SCons configuration files are still in bold face, but the built files are in italics.

and Fortran compilers; we can also set any other construction variables we want to use explicitly.

The variable name `env` to which this construction environment is assigned is then exported to the subsidiary configuration files by calling the `Export()` function. The `env` construction environment is now available for import by the subsidiary configuration files `libx/SConscript` and `progy/SConscript`, which we call by using the `SConscript()` function.

Our library subdirectory contains three source files. We want one of them, `file2.c`, to be built with a `-DXYZZY` command-line option:

```
# Subsidiary SConscript file for
# building libx.
Import('env')
file2_o = env.Object('file2.c',
                     CPPDEFINES = ['XYZZY'])
env.StaticLibrary('x', ['file1.c', file2_o,
                        'file3.c'])
```

The first `Import()` line imports the `env` construction environment, which is then used to build the object files and the library. The second line shows an explicit build of an object file, which is necessary here because we want this object file built with the desired `DXYZZY` option. Notice that we set this using the platform-independent `CPPDEFINES` variable, not `CCFLAGS`, so that SCons could construct the right command-line options when run on Windows. The second line also saves the return value from the `env.Object()` builder call. The returned value, which we save as the `file2_o` variable, is a list containing a *node*, an internal SCons object that represents the file that will be built from `file2.c`. Notice that we use this node as *source* for the `env.StaticLibrary()` call on the third line. The advantage, again, is that our build configuration is now platform-independent: SCons will remember whether the built object file is actually called `file2.o` (on Linux or Unix) or `file2.obj` (on Windows).

Notice also that all file names are interpreted relative to the directory in which the `SConscript` file lives, even though SCons will execute the commands from the top-level directory in which the `SConstruct` file lives.

In our other subdirectory, we want to compile the three source files into a program that we link with the library from the first subdirectory. We want to build all three source files with `-DFOO` and `-DBAR` command-line options, but we also want to build one of the three files with the `-Wall` option. (Because this is a compiler-specific option, it makes our configuration nonportable to compilers other than `gcc`.) The `SConscript` file looks like this:

```
# Subsidiary SConscript file for
# building progy.
Import('env')
env = env.Copy(CPPDEFINES = ['FOO', 'BAR'],
               LIBS = ['x'],
               LIBPATH = ['../libx'])
file5_o = env.Object('file5.c',
                     CCFLAGS = '-Wall')
env.Program('progy', ['file4.c', file5_o,
                      'file6.c'])
```

Because we want to build all the object files with the same options, we make a copy of the imported construction environment by calling `env.Copy()` and specifying the variables we want to use in this file. We can still override specific variables when calling a builder, like we do here when we set `CCFLAGS` when calling `env.Object()` to compile the object file for `file5.c`. We then use the same technique of using the returned value from the `env.Object()` call as an input source file when we call `env.Program()`. We link against the library in the first subdirectory by setting the `LIBS` and `LIBPATH` variables in the construction environment. We could have set these directly when calling `env.Program()`, but it was more convenient to do so when copying the construction environment, which would let us link other programs in this directory with the same library. We only specify the base name of the library we built: we don't need to specify the `lib` prefix or the `.a` suffix.

When we build this example by calling `scons`, the object files, library, and program are all built in their respective directories. Figure 2 shows the resulting tree.

Building Variants

Another common complicating factor in modern software builds is the need to build multiple variants of the software. It might be necessary, for example, to build a debug version of a program during normal development, and then build an optimized version of the same code base when building the software for official release.

SCons makes this easy by providing access to command-line arguments that can be consulted to change the way in which we set variables in our construction environment. Suppose we want to let a user build our example project with a command-line argument of `DEBUG=1` to specify that a debug version should be built and `OPT=1` to specify that an optimized version should built. We can do this by using the `ARGUMENTS` dictionary, which SCons provides to hold the values of any command-line arguments like these. Our `SConstruct` file now looks like this:

Another common complicating factor in modern software builds is the need to build multiple variants of the software.

```
# Top-level SConstruct file that calls
# subsidiary SConscript files to build a
# library and a program, checking for
# command-line settings of DEBUG= and OPT=.
env = Environment(CC = 'gcc'
                  FORTRAN = 'g77')
if int(ARGUMENTS.get('DEBUG', 0)):
    env.Append(CCFLAGS = '-g',
               CPPDEFINES = ['DEBUG'])
if int(ARGUMENTS.get('OPT', 0)):
    env.Append(CCFLAGS = '-O')
Export('env')
SConscript('libx/SConscript')
SConscript('progy/SConscript')
```

We used some Python code here: the `ARGUMENTS.get()` method fetches the specified value for the argument, `DEBUG` or `OPT`. If a value isn't specified on the command line, the `ARGUMENTS` dictionary won't have a value for that keyword, so the second argument to the `ARGUMENTS.get()` method is returned as the default value. In our example, both `DEBUG` and `OPT` have default values of `0`. We then wrap the calls to `ARGUMENTS.get()` in the Python `int()` function, which returns the integer value for the string. (Without the call to `int()`, a string value of `"0"` on the command line would actually evaluate true.)

We must now modify our subsidiary SConscript slightly to accommodate the possibility that the top-level SConstruct file could set the `CCFLAGS` or `CPPDEFINES` construction variables. Recall that we built the `libx/file2.c` file with a `CPPDEFINES` value of `XYZZY` and the `progy/file5.c` file with a `CCFLAGS` value of `-Wall`. If we didn't make any changes, those values would overwrite the `-g`, `-O`, or `DEBUG` values set by the top-level `SConstruct` file. We can accommodate this by modifying the `libx/SConscript` file to this:

```
# Subsidiary SConscript file for building
# "libx" without overriding an already-set
# CPPDEFINES value.
Import('env')
env2 = env.Copy()
env2.Append(CPPDEFINES = ['XYZZY'])
file2_o = env2.Object('file2.c')
env.StaticLibrary('x', ['file1.c', file2_o,
                        'file3.c'])
```

The new idiom here is to call the `env.Append()` method to append the `CPPDEFINES` value to the value (if any) set by the top-level `SConstruct` file. Similarly, we modify the `progy/SConscript` file as follows:

```
# Subsidiary SConscript file for building
# "progy" without overriding already-set
# CPPDEFINES or CCFLAGS values.
Import("env")
env = env.Copy(LIBS = ['x'],
               LIBPATH = ['../libx'])
env.Append(CPPDEFINES=['FOO', 'BAR'])
file5_o = env.Object('file5.c',
              CCFLAGS = '$CCFLAGS -Wall')
env.Program('progy', ['file4.c', file5_o,
                      'file6.c'])
```

Notice here that in addition to an `env.Append()` call for the `CPPDEFINES` values, we've had SCons expand the existing `$CCFLAGS` value in the new `CCFLAGS` value for compiling the `file5.c` file.

One subtle way in which this SCons configuration is an improvement over similar Make-based schemes is that the resulting object files and program not only depend on the source files, but also on the command-line options used to build them. This means that the first time we specify `DEBUG=1` on the command line, SCons will realize that none of the already built object files and programs will have been built that way, so it'll rebuild them all for us, correctly, without having to explicitly remove them first. This avoids a whole class of problems in Make-based builds in which object files built with different options can get linked together, causing maddeningly obscure bugs that are extremely difficult and time-consuming to track down.

```
├──SConstruct
├──libx/
│    ├──SConscript
│    ├──file1.c
│    ├──file2.c
│    └──file3.c
├──progy/
│    ├──SConscript
│    ├──file4.c
│    ├──file5.c
│    └──file6.c
├──debug/
│    ├──libx/
│    │    ├──file1.o
│    │    ├──file2.o
│    │    ├──file3.o
│    │    └──libx.a
│    └──progy/
│         ├──file4.c
│         ├──file5.c
│         ├──file6.c
│         └──progy
└──opt/
     ├──libx/
     │    ├──file1.o
     │    ├──file2.o
     │    ├──file3.o
     │    └──libx.a
     └──progy/
          ├──file4.c
          ├──file5.c
          ├──file6.c
          └──progy
```

Figure 3. Mirrored build trees for side-by-side debug and optimized versions of the same program. The SCons configuration files are still in bold face, and built files and directories are in italics, including the `debug` and `opt` subdirectories containing the different built versions.

Building Multiple Variants Side-by-Side

An alternative approach to building software in variant forms is to build multiple variants in side-by-side directories. This might be useful if we want to be able to compare the behavior of our debug and optimized versions (perhaps enabling debugging has an unintended side effect in the program), or if we want to build a program for multiple platforms in our network using the same NFS-mounted source tree everywhere.

SCons makes this very easy through a `build_dir` argument to the `SConscript()` function. This argument tells SCons where the target files defined by the builder calls in the specified SConscript file should be placed. The `build_dir` argument is usually accompanied by a `duplicate=0` argument that tells SCons to *not* duplicate the source files in the specified build directory as well. (By default, SCons would duplicate these source files to guarantee a correct build in unusual end cases involving generated header files and the use of the C/C++ `#include` directive with file names in double quotes rather than angle brackets.)

In our example, the top-level SConscript file will use these arguments to specify that the source files in our two source directories should each be built in side-by-side debug and optimized versions underneath subdirectories called `debug` and `opt`, respectively:

```
# Top-level SConstruct file that calls
# subsidiary SConscript files to build a
# library and a program, building side-by-
# side variants. The variant in the debug/
# subdirectory is built with -g -DDEBUG,
# and the variant in the opt/ subdirectory
# is built with -O.

env = Environment(CC = 'gcc',
                  FORTRAN = 'g77',
                  CCFLAGS = '-O')

Export('env')
SConscript('libx/SConscript',
           build_dir='opt/libx',
           duplicate=0)
SConscript('progy/SConscript',
           build_dir='opt/progy',
           duplicate=0)

env = env.Copy(CCFLAGS = '-g',
               CPPDEFINES=['DEBUG'])

Export('env')
SConscript('libx/SConscript',
           build_dir='debug/libx',
           duplicate=0)
SConscript('progy/SConscript',
           build_dir='debug/progy',
           duplicate=0)
```

Chez Thiruvathukal

Namespace Thiruvathukal

What to name our new little variable? Since he was born the week before the election, I found myself referring to him during diaper changes as "The Gentleman from Illinois." It fit: he's already well-known for loud filibusters. Still, he did need a proper name.

My friends suggested GTK 2.0, G++, G#, and *G. They say that once you have a child, you get an entire collection of new friends. (Apparently my friends are dyslexic, too, mistaking my initials for the Gnome Tool Kit).

Anyway, I wish to assure my readers that the boy has not been named after a programming language, library, or some linear combination thereof. Instead we gave him an Indian name, Rohan, which means "ascending" (www.babynamesindia.com/r.html) or "sandalwood" (www.babynamesworld.com/profile.php?seostats=1&name=Rohan).

Needless to say, my time these past several weeks has been dedicated to new-fatherhood tasks. In this installment, the kitchen is not as well stocked as usual, so I'll be preparing light fare using some rarely used ingredients that have been sitting in the refrigerator and spice rack for some time.

Fortran

Occasionally, I reminisce on the glory days of programming. You know, the days when programmers were real programmers (ok, cowboys/cowgirls). I was an undergraduate intern at Argonne National Laboratory in the physics division. I still vividly recall working with a scientist who suggested I add some statistical and reporting features to a Fortran code that numbered several hundred thousand lines and, when printed, amounted to approximately one standard ream of paper. To this day, thinking of this code reminds me of how it all got started. I wondered what it would look like if it had been written in C with its proper support for data structures and more modular constructs. At the time, OOP was more or less making its debut, but those using Fortran were still trying to enter the world of data abstraction and procedural programming.

One of my readers suggested I include a few words about modern Fortran implementations, such as Fortran 95, in my column. There are several choices out there: visit dmoz.org (an interesting project in its own right, it supports open directories on the Web, www.dmoz.org/Computers/Programming/Languages/Fortran/Compilers/). The g95 project (www.g95.org) is an open-source Fortran 95 implementation that supports most major platforms (Windows via Cygwin, Linux, and Mac OS X). Intel also provides a free Fortran implementation (www.intel.com/software/products/compilers/flin/). IBM and the Portland Group have some well-established commercial offerings. For those seeking a 64-bit offer-

We've accomplished this by assigning different construction environments to the `env` variable. The first copy of `env` builds all the files with a `-DDEBUG` option, and the second builds all the source files with a `-DOPT` option. By exporting different (but related) construction environments as `env` each time we call the `SConscript()` function, the subsidiary SConscript files that we "call" in this way don't have to have any special knowledge about whether their source files were built for a debug or optimized build. They simply `Import("env")` as they normally would, and SCons takes care of the rest.

When we build our example using this technique, SCons creates mirrored build trees under the `debug/` and `opt/` subdirectories, each in turn containing a `libx/` and `progy/` subdirectory containing the object files and library or program built with the appropriate debug or optimization flags (see Figure 3).

Building a Third-Party Library

Typically, a third-party library is prepared to be built with Make. Posix systems, including Linux and Unix systems, also typically have a configuration script (usually, but not always, called `configure`) that must be run to tailor the build to a particular system.

The simplest way to do this is to have SCons call Make to build the library. Even if SCons configuration files are easier to read than Make files, there's little sense in rewriting all the work the library's authors put into making their software build.

The simplest way to have SCons call Make is to use the `Command()` builder. The `Command()` builder takes target file and source file arguments, like the other builders we've already seen, but it also takes as a third argument an explicit command or list of commands executed to build the target file.

ing, there's Open64 Fortran, which is derived from the SGI commercial Fortran implementation.

It's refreshing to see that Fortran, like Elvis, lives. Digging a bit deeper, I decided to check for the availability of Cobol implementations. SourceForge has two projects: Tiny Cobol (http://tiny-cobol.sourceforge.net) and Open Cobol (http://open-cobol.sourceforge.net). Although preliminary, these are key opportunities for supercharging Fortran applications with the robust reporting capabilities found in Cobol.

Major Java Update

The Java programming language has been refreshed to version 5.0 (also known as 1.5.0). I continue to be amused at the software industry's obsession with version numbers. Microsoft was really onto something when it moved to version numbering based on year/edition.

In any event, don't let the version numbering confuse you. This Java update is what many of us have been waiting for, but it doesn't completely address what those of us in high-performance and scientific computing had been hoping for.

The major new language features include generics, the original term used to address parameterized class types (such as `List<int>` or `Map<string, Employee>`). Parameterized classes are highly useful and take much of the pain out of object-oriented programming.

Some other new features (imported from Microsoft C#) are boxing and autoboxing, which allow primitive types (`int`, `float`, `double`) to be assigned to and from object types. In the past, Java programmers had to use so-called wrapper classes to assign integers to objects (`[Object x = new Integer(5);]`) or to go in the reverse direction (`int [xValue = x.intValue()]`). Needless to say, the marriage between primitive and object types in Java was one of inconvenience—until now. The same code can now be written as `[Object x = 5]` and `[int xValue = (int) x]`.

Python (which by now everyone knows is one of my favorite languages after Java) provides support for natural iteration. You can write code like this to examine all elements of a list:

```
myList = [1, 2, 3]:
for item in myList:
   print item
```

Java now provides an enhanced `for` loop, which allows you to do something similar. It's still much more concise in Python, but here's the equivalent code:

```
ArrayList<Integer> myList = new
ArrayList<Integer>();
myList.add(1);
myList.add(2);
myList.add(3);

for (Integer item : myList)
    System.out.println(item);
```

Although such new features make programming in Java much more pleasant, there are still some things missing, such as support for true arrays (notably, rectangular arrays). Hopefully, Sun will wake up and smell the Java. We need proper arrays, operators, and support for complex primitives to write scientific and engineering codes in Java.

Nevertheless, these new features are a step in the right direction. Similar to American politics, these changes are more likely to affect the Java base than the specialized uses our community needs. But the changes *will* have impact. Perhaps 18 years from now, similar to Fortran 77, we'll be working with Java 2022.

As a specific example, suppose we want to build a recent stable version of the Atlas library from its downloadable archive file. Unpacking the archive shows us that it unpacks itself into an `ATLAS/` subdirectory. According to the `INSTALL.txt` file contained in the Atlas distribution, we need to execute the commands `"make config"` and `"make install arch=arch"` to build Atlas. The file also tells us that we can pass arguments to the `"make config"` command that specify our C and Fortran compilers. From having built the library before, we see that the full build will create five libraries in the `lib/arch/` subdirectory. Putting this all together, our SConscript file looks like this:

```
# Subsidiary SConscript file for building
# ATLAS from its downloadable .tar.gz
# archive file.

ATLASVERSION = ARGUMENTS.get(ATLASVERSION,
                             '3.6.0')
ATLASARCH = ARGUMENTS.get(ATLASARCH,
                       'Linux_PIIISSE1')

env = Environment(CC = 'gcc',
        FORTRAN = 'g77',
        ATLASVERSION = ATLASVERSION,
        ATLASARCH = ATLASARCH,
        ATLAS_LIB_DIR = 'ATLAS/lib/$ARCH')

ATLAS_libraries = Split("""
    $ATLAS_LIB_DIR/libatlas.a
    $ATLAS_LIB_DIR/libcblas.a
    $ATLAS_LIB_DIR/libf77blas.a
    $ATLAS_LIB_DIR/liblapack.a
    $ATLAS_LIB_DIR/libtstatlas.a
""")

env.Command(ATLAS_libraries,

            'atlas${ATLASVERSION}.tar.gz',
    [
        '$TAR zxf $SOURCE',
        'cd ATLAS && ' + \
  'make config CC=$CC F77=$FORTRAN',
        'cd ATLAS && ' + \
  'make install arch=$ATLASARCH',
    ])
```

The first section uses the `ARGUMENTS.get()` method to let the user specify on the command line an explicit version of the Atlas library to build, or an explicit architecture to build for. The default values will build version `3.6.0` (the latest stable release) for Linux (on an older Pentium III system), but the user could build development version `3.7.8` by specifying `ATLASVERSION` on the command line as follows:

```
  $ scons ATLASVERSION=3.7.8
```

The second section creates the construction environment that we'll use to build Atlas. We've chosen to specify explicitly our C and Fortran compilers, although we could have used the default compilers that SCons would pick for us. We've set construction variables called `ATLASVERSION` and `ATLASARCH` to the values of the same-named Python variables we let the user set. We also created an `ATLAS_LIB_DIR` construction variable to define the path where Atlas builds its libraries.

The third section defines the five libraries that we expect will be created by this build of Atlas. We tell SCons to use the specified `$ATLAS_LIB_DIR` variable, which when expanded indicates that the libraries will be found (by default) in the `ATLAS/lib/Linux_PIIISSE1` subdirectory. (`$ATLAS_LIB_DIR` doesn't actually get expanded until SCons determines that it needs to build the libraries.) We could have hard-coded this as a Python list, but rather than quote each library name individually, we used the SCons-supplied `Split()` function to split the multiline string within the Python triple-quote syntax into the separate library names.

The last section is where we call the `Command()` builder to tell SCons how to make the libraries. The `ATLAS_libraries` Python variable, containing the list of libraries to be built, is the target (the first argument), and the source (the second argument) is the current Atlas distribution file, which we expect to find in the same directory as the SConscript file. If it were actually located elsewhere, we could simply specify a relative pathname (`../downloads/atlas${VERSION}.tar.gz`) or absolute pathname (`/usr/local/downloads/atlas${VERSION}.tar.gz`) to the location.

The first command unpacks the archive; SCons sets the `$TAR` variable to an appropriate utility, and the `$SOURCE` variable refers to the source archive, so we don't have to specify the location more than once. The second and third commands execute the appropriate commands to build and install Atlas. Because the archive unpacks into an `ATLAS` subdirectory, we must add an explicit `"cd ATLAS"` to the beginning of each command so that the `make` commands are executed from within the proper, unpacked directory.

If we added the above code directly to our SConstruct file, we'd see the source tree in Figure 4, which contains the Atlas archive and the build subdirectory abridged to show just the files we're interested in.

This seems like a lot of work: we had to unpack the library to examine its build and installation instructions, figure out what directory it unpacks into, and then examine an already built copy to figure out what library files it creates. But the advantage of taking the time to gather this information and put it into an SConscript file is that it will now become much simpler to keep our software in sync with future releases and other versions of Atlas. All we'll need to do is download the next release and change the version number in our SConscript file from `3.6.0` to `3.7.8`, for example. The only time we would need to look any deeper would be if the Atlas developers substantially changed their build and installation procedures, or if a new version of Atlas added another built library.

Linking with the Built Library

Having built Atlas, we presumably want to use it in the rest of our software. Recall that we do this by setting two SCons construction variables: `LIBS` tells SCons which libraries to link the software with, and `LIBPATH` tells SCons where to find the libraries. It's most convenient to add these to the same construction environment in which we set the `ATLAS_LIB_DIR` construction variable:

```
env = Environment(CC = 'gcc',
         FORTRAN = 'g77',
ARCH = 'Linux_PIIISSE1',
ATLAS_LIB_DIR = 'ATLAS/lib/$ARCH',
LIBPATH = ["$ATLAS_LIB_DIR"],
LIBS = ['atlas', 'lapack'])
```

Believe it or not, that's all we need to do. SCons will now add the appropriate options when linking any program specified by the `env.Program()` builder called through this construction environment.

Advanced Usage: Avoiding Build Command Duplication

If we have several third-party packages we want to build with the same set of commands, it would be tedious and error-prone to repeat those commands for each and every package. Recall that SCons helps avoid duplication in our build configuration by letting us collect variables into a construction environment. Similarly, SCons also lets us define our own builder object to collect common lists of commands for reuse in building multiple output files without duplication.

Suppose, for example, that we have many packages to configure using the same "`configure-make-make install`" list of commands (the traditional package-building recipe for packages generated using the GNU `Autoconf` utility). We could do this in SCons as follows:

```
MakeInstallBuilder = Builder(
       action = [
         '$TAR xf $SOURCE',
         'cd ${TARGET.dir} && ' + \
             './configure -prefix=$PREFIX',
         'cd ${TARGET.dir} && make',
         'cd ${TARGET.dir} && make install',
       ])

env = Environment()
env['BUILDERS']['MakeInstall'] = \
                         MakeInstallBuilder

env.MakeInstall('/usr/local/bin/foo',
                'foo-1.0.tar.gz',
                PREFIX='/usr/local')
env.MakeInstall('/usr/local/bin/bar',
                'bar-0.9.tar.gz',
                PREFIX='${TARGET.dir.dir}')
```

The first section defines the `Builder` object, which we assign to a Python variable called `MakeInstallBuilder`.

```
src/
  ├──SConstruct
  ├──atlas3.6.0.tar.gz
  ├──ATLAS
  │    ├──lib/
  │    │    └──Linux PIIISE1
  │    │            ├──libatlas.a
  │    │            ├──libcblas.a
  │    ┆            ├──libf77blas.a
  │    ┆            ├──liblapack.a
  │                 └──libtstatlas.a
  ├──libx/
  │    ├──SConscript
  │    ├──file1.c
  │    ├──file2.c
  │    ├──file3.c
  │    ├──file1.o
  │    ├──file2.o
  │    ├──file3.o
  │    └──libx.a
  └──progy/
       ├──SConscript
       ├──file4.c
       ├──file5.c
       ├──file6.c
       ├──file4.c
       ├──file5.c
       ├──file6.c
       └──progy
```

Figure 4. Source tree after building the Atlas library from code added to the `SConstruct` file. The `ATLAS` subdirectory has been created by unpacking the `atlas3.6.0.tar.gz` file; the libraries were created by having SCons execute the necessary build commands.

The `action` keyword argument is required because it tells SCons what commands to invoke when building target files with this builder. The `$`-variables in the command lines do *not* get expanded when we define the builder; rather, they get expanded when the builder is invoked to actually build any necessary target files. This lets us pass a value to the `–prefix=` argument to be specified later, when we actually call the builder.

The second section shows us the attachment of the builder to a construction environment; now we can call it to arrange for target files to be built. We can specify the name we want to use to call the builder as the value in the '`BUILDERS`' dictionary to which we assign the builder—in this case, the name is `MakeInstall`.

In the last section, we call the builder twice to build and install two different packages. In the first call, we explicitly set the `PREFIX` value to `/usr/local`, reflecting where we want the package installed. Although it's clear where we want the installation, it has the slight disadvantage of requiring the prefix to be changed in two places if we ever want to do that. In the second call, we generalized the `PREFIX` value by using the `dir` attribute of the SCons `TARGET` variable. This specifies that we want to insert the directory under which the `$TARGET` file can be found; by using it twice, we essentially "back up" to the `/usr/local` directory without having to specify it by hand. This is a little less obvious, but it means that we're specifying the `PREFIX` in only one place.

This article, of course, only scratches the surface of what SCons can do. Additional key SCons features not covered here include support for Java and SWIG, for building TeX and LaTex documents, and building project and solution files for Visual Studio versions 6 and 7 (.NET). SCons also has integrated support for multiplatform configuration (similar to the GNU `Autoconf` utility), can build multiple targets in parallel using a `-j` command-line option, and can build software from central repositories of source or target files.

SCons is in active development by a team of volunteer developers led by me. New releases appear approximately quarterly, and are developed using a methodology that makes extensive use of regression tests to ensure that new releases don't break existing functionality.

Upcoming areas of work on SCons include integrated support for distributed builds and remote execution of build commands, support for Visual Studio version 8, and scaling the performance and memory usage of SCons to extremely large projects.

Steven Knight has worked as a software engineer, executive, and consultant for more than 20 years. In 2000, his design for a next-generation build tool won the Software Carpentry contest and served as the basis for the SCons Project, which he founded the following year. He is currently making use and enhancement of SCons to simplify software builds the focus of his career. Contact him at knight@scons.org.

Submissions: Send one PDF copy of articles and/or proposals to Norman Chonacky, Editor in Chief, norman.chonacky@yale.edu. 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 2005: $42 for Computer Society members (print only) and $42 for AIP society members (print plus online). For more information on other subscription prices, see www.computer.org/subscribe or https://www.aip.org/forms/journal_catalog/order_form_fs.html. Computer Society back issues cost $20 for members, $96 for nonmembers; AIP back issues cost $22 for members.

Postmaster: Send undelivered copies and address changes to *Computing in Science & Engineering*, 445 Hoes Ln., Piscataway, NJ 08855. Periodicals postage paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Corporation (Canadian distribution) publications mail agreement number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8 Canada. Printed in the USA.