SCons Design and Implementation

Steven Knight

Table of Contents

`SCons` is a software construction tool (build tool, or make tool) implemented in Python, which uses Python scripts as "configuration files" for software builds. Based on the design which won the Software Carpentry build tool competition, `SCons` solves a number of problems associated with other build tools, especially including the classic and ubiquitous `Make` itself.

Distinctive features of `SCons` include: a modular design that lends itself to being embedded in other applications; a global view of all dependencies in the source tree; an improved model for parallel (`-j`) builds; automatic scanning of files for dependencies; use of MD5 signatures for deciding whether a file is up-to-date; use of traditional file timestamps instead of MD5 signatures available as an option; use of Python functions or objects to build target files; easy user extensibility.

This paper discusses the goals of the `SCons` project, gives an overview of the design of `SCons` itself, describes the development process used, and discusses future plans and directions for the tool.

Introduction

More than twenty years after its creation, the classic UNIX `Make` utility and its descendants are still the dominant way in which software is built. `Make` has maintained this position despite the fact that the intervening years have revealed many shortcomings of the `Make` model for building software:

- The use of timestamps to decide when a file has been updated is imprecise and prone to error, especially across distributed file systems such as NFS.
- Builds of typical large software systems still take hours, if not days, despite the tremendous advances in CPU and disk speeds over recent years.
- `Make` maintains static definitions of dependencies in its `Makefiles`. Much effort has been put into utilities (`mkdepend`, `gcc -M`) and schemes (`Makefile.d` files) to try to keep `Makefile` dependencies up-to-date, but these only confirm that `Make`'s static dependencies are inherently fragile.
- The standard recursive use of `Make` for build hierarchies leads to incomplete dependency graphs, which must be overcome by manually changing the order in which directories are built, or through the use of multiple build passes.

One need only look at the plethora of helper and wrapper utilities (automake, easymake, imake, jmake, makeLib, maketool, mkmed, shake, SMake, TMAKE) and complete alternatives to `Make` (Ant, bake, bau, bras, Cake, Cons, Cook, Jam, jmk, jus, makeme, mash, MK, nmake, Odin, VMake) that have been created over the years to realize that vanilla `Make` is not satisfying everyone's build requirements. So why Yet Another build tool?

Enter Software Carpentry

Most of the build tools just mentioned were written by programmers and for programmers. The fact that most programmer-friendly utilities do a poor job of fulfilling the needs of non-programmers prompted Greg Wilson to organize the Software Carpentry competition in January 2000. Software Carpentry was an open design contest with the express goal of producing a set of next-generation utilities, including a build tool, that would be accessible not only to programmers but also to computer *users* such as physical scientists.

The key to this usability would be that all of these utilities, including the build tool, would be written in Python. This provided the catalyst for actually pursuing an idea that had been floating around one of the more intriguing `Make` alternatives, a Perl utility called `Cons`. What if the friendlier syntax of Python could be married to the architectural advantages of `Cons`?

The resulting merged design, at that time named `ScCons`, won the Software Carpentry build tool competition. CodeSourcery (by then the administrators of the competition) ultimately decided not to fund development of the build tool, but the seed had been planted and the design had taken root.

Cons

It helps to know something about `Cons`. `Cons` was first released in 1996 by Bob Sidebotham, then an employee of Fore Systems, and it has a number of distinctive features that set it apart from most `Make`-alikes:

- `Cons` "configuration files" are not Yet Another invented mini-language, but are actually *Perl scripts*, which means the full power and flexibility of a real scripting language can be applied to build problems.
- `Cons` builds everything from a single process at the top of the source tree, with a global view of the dependencies.
- `Cons` scans files automatically for dependencies such as files specified on `#include` lines.
- `Cons` decides if a file was out-of-date by using MD5 checksums of the contents of files, not timestamps.

Despite all of these intriguing architectural features, the great strength of `Cons`—being written in Perl—was also one of its weaknesses, turning away many potential users due to the (real or perceived) steep learning curve of Perl.

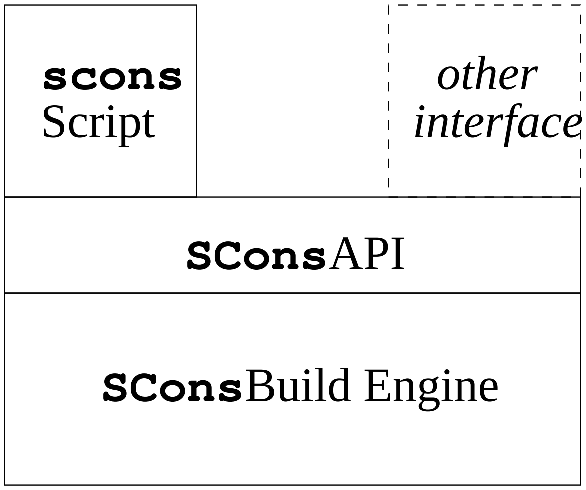
SCons

Through the `ScCons` contest entry, `SCons` is the direct descendant of the `Cons` architecture, and is currently under active, supported development with a growing body of users. Its first release was 13 December 2001, under the simple and non-restrictive MIT license, and from the outset, the goal of the members of the `SCons` project has been to deliver a stable, reliable tool that can be used for industrial-strength software builds.

The rest of this paper will give an overview of the `SCons` design (including its architecture and interface), describe the development methodology used, and discuss future directions for `SCons`.

Architecture

The `SCons` architecture consists of three layers:



- The SCons *Build Engine*, a package of Python modules that handle dependency management and updating out-of-date objects.
- The SCons *API* (applications programming interface) between the Build Engine and the user interface.
- The scons *script* itself (note lower case *sc*), which is the pre-provided interface to the Build Engine.

Notice that this architecture separates the internal workings of SCons (the Build Engine) from the external user interface. The benefit is that the SCons Build Engine can be imported into any other software package written in Python to support a variety of user interfaces—or, to look at it in reverse, other software interfaces can use the SCons Build Engine to manage dependencies between their objects.

Because the SCons package itself is modular, only those parts of the package relevant to the embedding interface need be imported; for example, a utility that wants to use only file timestamps for checking whether a file is up-to-date need not import the MD5 signature module.

The SCons Build Engine

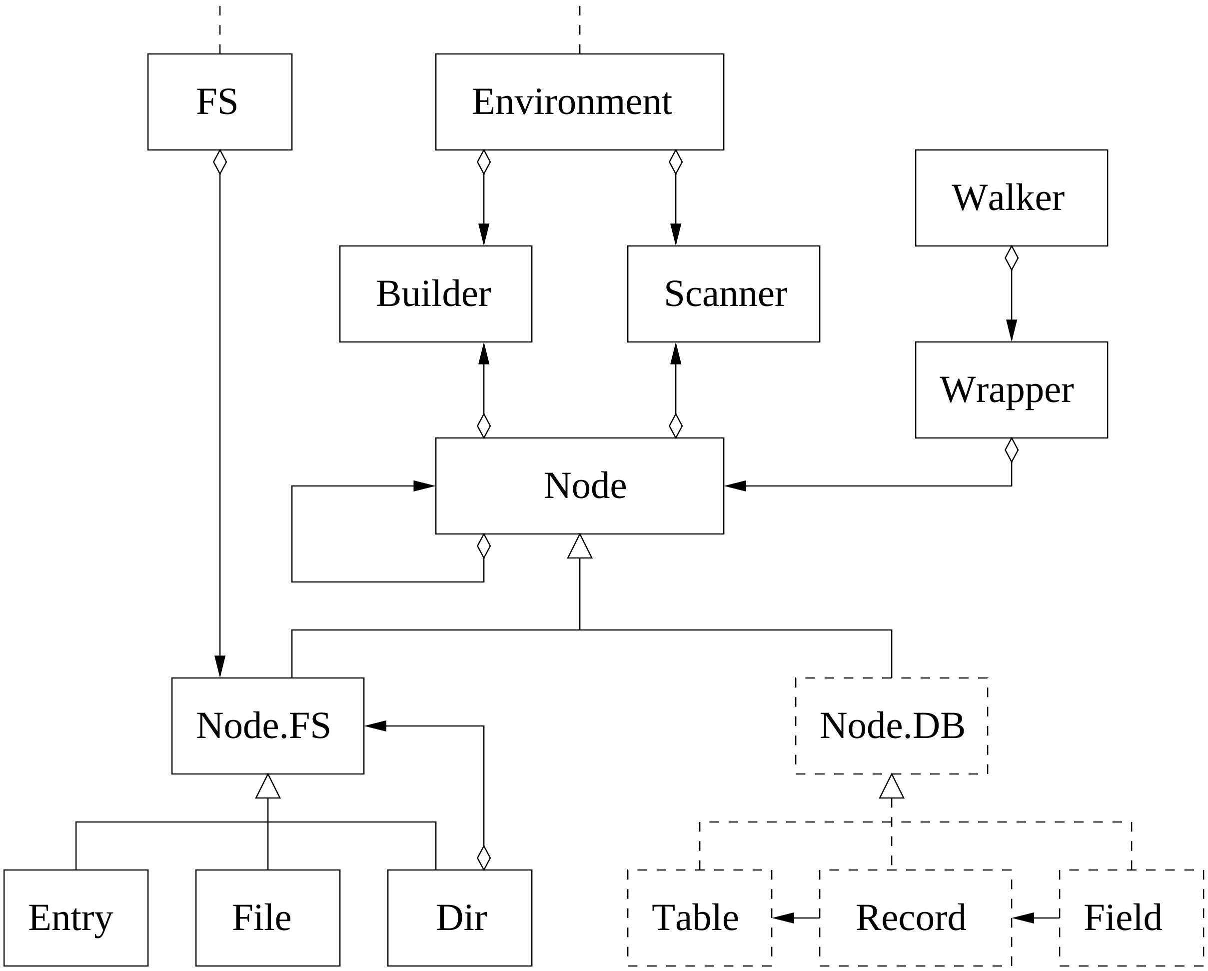
The Build Engine is a package of Python modules that form the heart of SCons. The Build Engine can be broadly divided into five architectural subsystems, each responsible for a crucial part of SCons functionality:

- A *node* subsystem, responsible for managing the files (or other objects) to be built, and the dependency relationships between them.
- A *scanner* subsystem, responsible for scanning various file types for implicit dependencies.
- A *signature* subsystem, responsible for deciding whether a given file (or other object) requires rebuilding.
- A *builder* subsystem, responsible for actually executing the necessary command or function to build a file (or other object).
- A *job/task* subsystem, responsible for handling parallelization of builds.

The rest of this section will provide a high-level overview of the class structure of each of these Build Engine subsystems.

Node Subsystem

The node subsystem of the Build Engine is responsible for managing the knowledge in `SCons` of the relationships among the external objects (files) it is responsible for updating. The most important of these relationships is the dependency relationship between various `Node` objects, which `SCons` uses to determine the order in which builds should be performed.



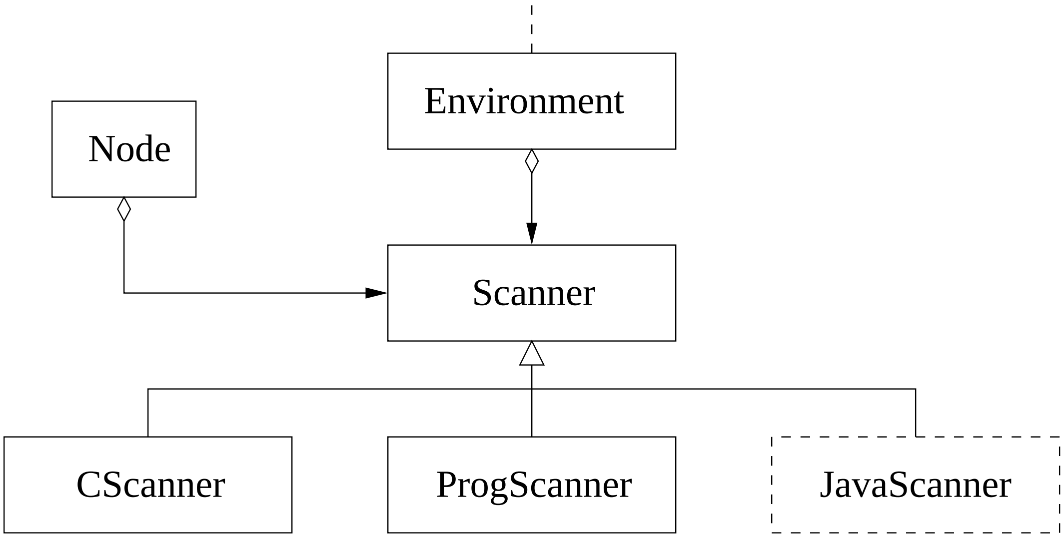
The `scons` script (or other user interface) tells the Build Engine about dependencies through its `construction environment` API. The Build Engine also discovers dependencies automatically through the use of `Scanner` objects.

Subclasses of the `Node` class maintain additional relationships that reflect the real-world existence of these objects. For example, the `Node.FS` subclass is responsible for managing a representation of the directory hierarchy of a file system.

A `Walker` class is used by other subsystems to walk the dependency tree maintained by the `Node` class. The `Walker` class maintains a stack of `Node` objects visited during its depth-first traversal of the dependency tree, and uses an intermediate node `Wrapper` class to maintain state information about a `Node` object's dependencies.

Scanner Subsystem

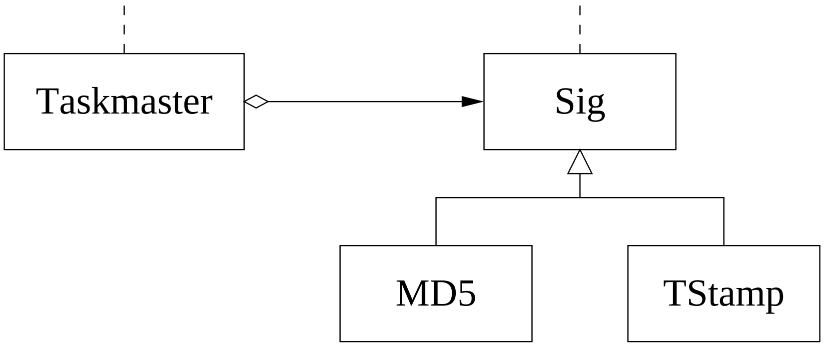
The scanner subsystem is responsible for maintaining objects that can scan the contents of a `Node`'s for implicit dependencies.



In practice, a given `Scanner` subclass object functions as a prototype, returning clones of itself depending on the `construction environment` values governing how the `Node` should be scanned.

Signature Subsystem

The signature subsystem is responsible for computing signature information for `Node` objects. The signature subsystem in `SCons` supports multiple ways to determine whether a `Node` is up-to-date by using an abstract `Sig` class as a strategy wrapper:



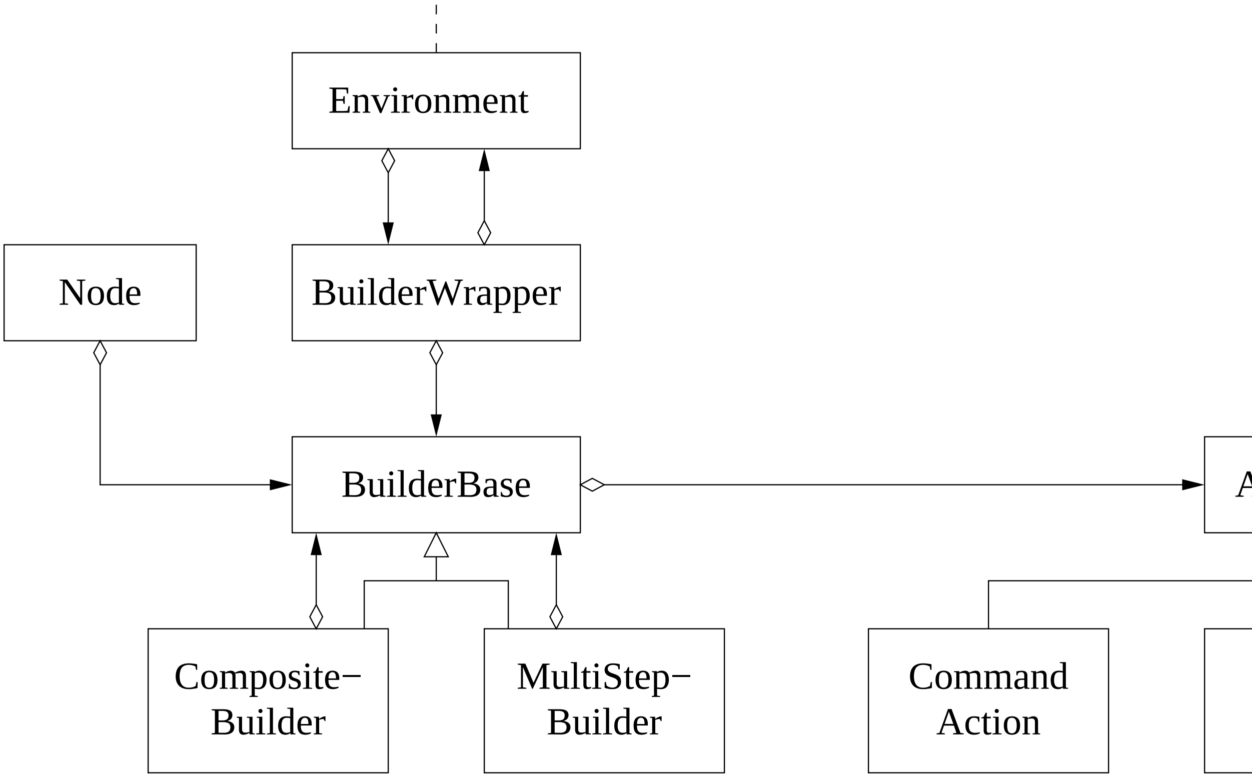
By default, `SCons` tracks dependencies by computing and maintaining MD5 signatures for the contents of each source file (or other object). The signature of a *derived* file consists of the aggregate of the signatures of all the source files *plus* the command-line string used to build the file. These signatures are stored in a `.sconsign` file in each directory.

If the contents of any of the source files changes, the change to its MD5 signature is propogated to the signature of the derived file(s). The simple fact that the new signature does not match the stored signature indicates that the derived file is not up to date and must be rebuilt.

A separate `TimeStamp` subclass of the `Sig` class supports the use of traditional file timestamps for deciding whether files are up-to-date.

Builder Subsystem

The SCons Build Engine records how out-of-date files (or other objects) should be rebuilt in Builder objects, maintained by the builder subsystem:

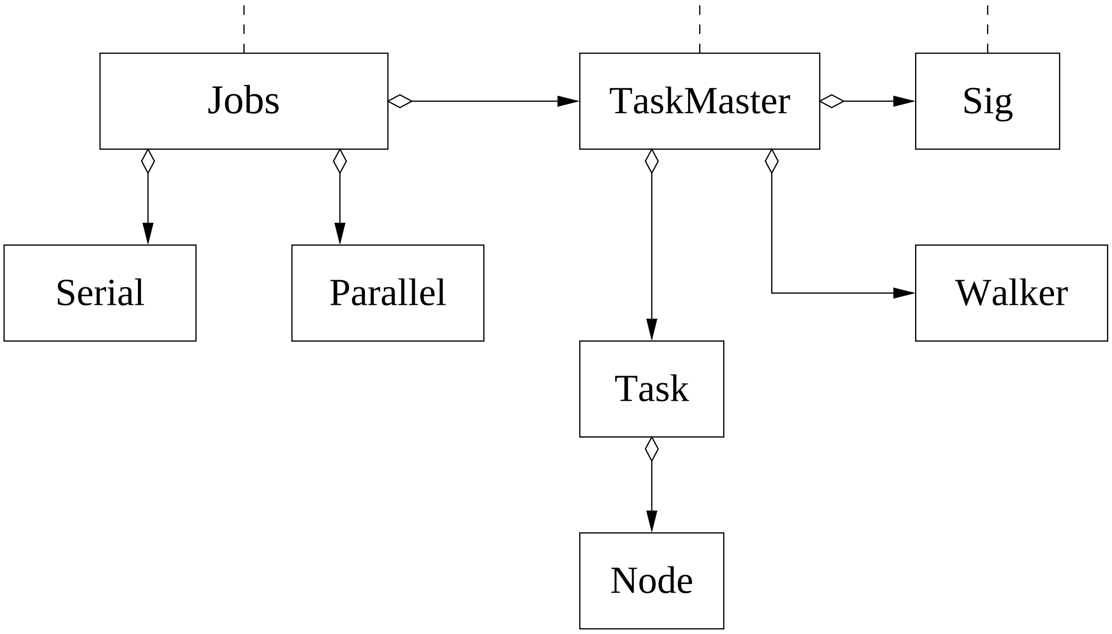


The actual underlying class name is BuilderBase, and there are subclasses that can encapsulate multiple Builder objects for special purposes. One subclass (CompositeBuilder) selects an appropriate encapsulated Builder based on the file suffix of the target object. The other (MultiStepBuilder). can chain together multiple Builder objects, for example, to build an executable program from a source file through an implicit intermediate object file.

A BuilderBase object has an associated ActionBase object responsible for actually executing the appropriate steps to update the target file. There are three subclasses, one for externally executable commands (CommandAction), one for Python functions (FunctionAction), and one for lists of multiple Action objects (ListAction).

Job/Task Subsystem

SCons supports parallel builds with a thread-based tasking model, managed by the job/task subsystem.



Instead of performing an outer-loop recursive descent of the dependency tree and then forking a task when it finds a file that needs updating, `SCons` starts as many threads as are requested, each thread managed by the `Jobs` class. As a performance optimization, the `Jobs` class maintains an internal distinction between `Serial` and `Parallel` build jobs, so that serial builds don't pay any performance penalty by using a multi-threaded implementation written for `Parallel` builds.

Each `Jobs` object, running in its own thread, then requests a `Task` from a central `Taskmaster`, which is responsible for handing out available `Task` objects for (re-)building out-of-date nodes. A condition variable makes sure that the `Jobs` objects query the `Taskmaster` one at a time.

The `Taskmaster` uses the node subsystem's `Walker` class to walk the dependency tree, and the `Sig` class to use the appropriate method of deciding if a `Node` is up-to-date.

This scheme has many advantages over the standard `Make` implementation of `-j`. Effective use of `-j` is difficult with the usual recursive use of Make, because the number of jobs started by `-j` multiply at each level of the source tree. This makes the actual number of jobs executed at any moment very dependent on the size and layout of the tree. `SCons`, in contrast, starts only as many jobs as are requested, and keeps them constantly busy (excepting jobs that block waiting for their dependency files to finish building).

The `SCons` API

This section provides an overview of the `SCons` interface. The complete interface specification is both more detailed and flexible than this overview.

Construction Variables

In `SCons`, a `construction environment` is an object through which an external interface (such as the `scons` script) communicates dependency information to the `SCons` Build Engine.

A construction environment is implemented as a dictionary containing:

- construction variables, string values that are substituted into command lines or used by builder functions;
- one or more `Builder` objects that can be invoked to update a file or other object;
- one or more `Scanner` objects that can be used to scan a file automatically for dependencies (such as files specified on `#include` lines).

`Construction environments` are instantiated as follows:

```
env = Environment()
env_debug = Environment(CCFLAGS = '-g')
```

`Builder` Objects

An `SCons` `Builder` object encapsulates information about how to build a specific type of file: an executable program, an object file, a library, etc. A `Builder` object is associated with a file through an associated `construction environment` method and later invoked to actually build the file. The `Builder` object will typically use construction variables (such as `CCFLAGS`, `LIBPATH`) to influence the specific build execution.

`Builder` objects are instantiated as follows:

```
bld = Builder(name = 'Program', action = "$CC -o $TARGET $SOURCES")
```

In the above example, the `action` is a command-line string in which the Build Engine will interpolate the values of construction variables before execution. The actual `action` specified, though, may be a function:

```
def update(dest):
    # [code to update the object]
    return 0

bld = Builder(name = 'Program', function = update)
```

Or a callable Python object (or class):

```
class class_a:
    def __call__(self, kw):
        # build the desired object
    return 0

builder = SCons.Builder.Builder(action = class_a())
```

A `Builder` object may have the `prefix` and `suffix` of its target file type specified as keyword arguments at instantiation. Additionally, the suffix of the *source files* used by this `Builder` to build its target files may be specified using the `src_suffix` keyword argument:

```
bld_lib = Builder(name = 'Library', action = "$AR r $TARGET $SOURCES",
                  prefix = 'lib', suffix = '.a', src_suffix = '.o')
```

The specified `prefix` and `suffix` will be appended to the name of any target file built by this `Builder` object, if they are not already part of the file name. The `src_suffix` is used by the `SCons` Build Engine to chain together multiple `Builder` objects to create, for example, a library from the original source files without having to specify the intermediate `.o` files.

`Builder` objects are associated with a `construction environment` through a `construction variable` named `BUILDERS`, a list of the `Builder` objects that will be available for execution through the `construction environment`:

```
env = Environment(BUILDERS = [ Object, Library, WebPage, Program ])
```

`Scanner` Objects

`Scanner` objects perform automatic checking for dependencies by scanning the contents of files. The canonical example is scanning a C source file or header file for files specified on `#include` lines.

A `Scanner` object is instantiated as follows:

```
def c_scan(contents):
     # scan contents of file
     return # list of files found

c_scanner = Scanner(name = 'CScan', function = c_scan,
                    argument = None,
                    skeys = ['.c', '.C', '.h', '.H')
```

The `skeys` argument specifies a list of file suffixes for file types that this `Scanner` knows how to scan.

`Scanner` objects are associated with a `construction environment` through a `construction variable` named `SCANNERS`, a list of the `Scanner` objects that will be available through the `construction environment`:

```
env = Environment(SCANNERS = [ CScan, M4Scan ])
```

For utilities that will build files with a variety of file suffixes, or which require unusual scanning rules, a `Scanner` object may be associated explicitly with a `Builder` object as follows:

```
def tool_scan(contents):
    # scan contents of file
    return # list of files found

tool_scanner = Scanner(name = 'TScan', function = tool_scan)

bld = Builder(name = 'Tool', scanner = tool_scanner)
```

`BuildDir`

`SCons` supports a flexible mechanism for building target files in a separate build directory from the source files. The `BuildDir` syntax is straightforward:

```
BuildDir(source = 'src', build = 'bld')
```

By default, source files are linked or copied into the build directory, because exactly replicating the source directory is sometimes necessary for certain combinations of use of `#include "..."` and `-I` search paths. An option exists to specify that only output files should be placed in the build directory:

```
BuildDir(source = 'src', build = 'bld', no_sources = 1)
```

Repository

SCons supports the ability to search a list of code repositories for source files and derived files. This works much like Make's VPATH feature, as implemented in recent versions of GNU Make. (The POSIX standard for Make specifies slightly different behavior for VPATH.) The syntax is:

```
Repository('/home/source/1.1', '/home/source/1.0')
```

A command-line -Y option exists to allow repositories to be specified on the command line, or in the SCONSFLAGS environment variable (not construction variable!). This avoids a chicken-and-egg situation and allows the top-level SConstruct file to be found in a repository as well.

Cache

SCons supports a way for developers to share derived files. Again, the syntax is straightforward:

```
Cache('/var/build.cache/i386')
```

Copies of any derived files built will be placed in the specified directory with their MD5 signature. If another build results in an out-of-date derived file with the same signature, the derived file will be copied from the cache instead of being rebuilt.

The scons Script

The scons script provides an interface that looks roughly equivalent to the classic Make utility—that is, execution from the command line, and dependency information read from configuration files.

The most noticeable difference between scons and Make, or most other build tools, is that the configuration files are actually Python scripts, generically called "SConscripts" (although the top-level "Makefile" is named SConstruct). Users do not have to learn a new language syntax, but instead configure dependency information by making direct calls to the Python API of the SCons Build Engine. Here is an example SConstruct file which builds a program in side-by-side normal and debug versions:

```
env = Environment()
debug = env.Copy(CCFLAGS = '-g')

source_files = ['f1.c', 'f2.c', 'f3.c']

env.Program(target = 'foo', sources = source_files)
debug.Program(target = 'foo-debug', sources = source_files)
```

Notice the fact that this file is a Python script, which allows us to define and re-use an array that lists the source files.

Because quoting individul strings in long lists of files can get tedious and error-prone, the SCons methods support a short-cut of listing multiple files in a single string, separated by white space. This would change the assignment in the above example to a more easily-readable:

```
source_files = 'f1.c f2.c f3.c'
```

The mechanism to establish hierarchical builds is to "include" any subsidiary configuration files in the build by listing them explicitly in a call to the `SConscript` function:

```
SConscript('src/SConscript', 'lib/SConscript')
```

By convention, configuration files in subdirectories are named `SConscript`.

The `scons` script has intentionally been made to look, from the outside, as much like `Make` as is practical. To this end, the `scons` script supports all of the same command-line options supported by GNU `Make`: `-f` FILE, `-j`, `-k`, `-s`, etc. For compatibility, `scons` ignores those GNU `Make` options that don't make sense for the `SCons` architecture, such as `-b`, `-m`, `-S`, and `-t`. The intention is that, given an equivalent `SConstruct` file for a `Makefile`, a user could use `SCons` as a drop-in replacement for `Make`. Additional command-line options are, where possible, taken from the Perl `Cons` utility on which the `SCons` design is based.

Installation

Initial installation of a new utility provides the first, lasting impression of how well the software is likely to perform. From the start, `SCons` has made clean installation a priority.

Version Control

Distributing an application like `SCons` that depends on a package normally found in a library poses a problem. If the `scons` script and the `SCons` Build Engine are installed separately, it could be easy to introduce a version mismatch between the Build Engine installed in `/usr/lib/python*/site-packages` and the `scons` script installed in `/usr/bin`. Such a mismatch could possible mean exceptions that prevent builds, or even worse, silently unreliable builds.

To reduce the possibility of a version mismatch, the `scons` script looks first for its imported modules in `/usr/lib/scons-{version}/`, then in `/usr/lib/scons/`, and then in the normal `PYTHONPATH` locations, including `/usr/lib/python*/site-packages`). Searching in a version-specific library directory first makes it convenient to install and use multiple side-by-side versions of `SCons`, which is sometimes important when verifying that a new version does not introduce any errors into the local build process. Searching next in an `SCons`-specific library directory makes it convenient for other software to find the `SCons` Build Engine without having to worry about installing separate copies for multiple versions of Python.

Packages

`SCons` is currently distributed in the following packages:

Table 1.

`scons-`*version*`.tar.gz`	The traditional `.tar.gz` file, installable by running `setup.py`.
`scons-`*version*`.noarch.rpm`	An RPM file for typical installation.
`scons-`*version*`_all.deb`	A Debian package.

`scons-`*version*`.win32.exe`	A Windows installer.
`scons-`*version*`.src.rpm`	A source RPM file.
`scons-src-`*version*`.tar.gz`	A tarball of the `SCons` source tree, including the full set of regression tests.

Like other software written in Python, `SCons` benefits greatly from the tremendous effort put into the `distutils` by Greg Ward and others. These take care of 90% of the work by making it almost trivial to generate the appropriate RPM files, Debian packages, and Windows installer.

Default Builder Objects

As part of the installation process, `SCons` runs a set of scripts that look for popular compilers and other tools and set up appropriate default `Builder` objects for the tools found. These `Builder` objects are then used to initialize the default `construction environment` values.

Default Scanner Objects

Additionally, `SCons` comes with a stock set of `Scanner` objects for the various file types that it supports out of the box. Any unusal `Scanner` objects required for a specific tool will be detected at installation time and associated with the appropriate `Builder` object for the tool.

Development Process

The `SCons` project has paid particular attention from day one to the development process. One of the first internal documents produced was a set of Developer's Guidelines to provide a loose framework for what we were trying to accomplish and how we would go about accomplishing it. These Guidelines cover things like:

- `SCons` will be written to Python version 1.5.2 (to ensure usability by a wide install base).
- How `SCons` is be tested: which infrastructure modules to use, what platforms to test on, etc.
- Expectations for developers (subscribe to the mailing list, encouraged to register at SourceForge).
- Brief outline of how to use the change management systems (Aegis and CVS) for `SCons` development;.

Establishing these guidelines up front had two purposes: 1) Demonstrate the seriousness of the project to anyone wondering about joining the effort; 2) Give potential developers an idea up front as to whether their development style would mesh with the rest of the project.

Aegis

One of the most important aspects of the `SCons` development process is the use of Peter Miller's Aegis change management system. I had been using Aegis for personal projects for several years, and found its development methodology vastly improved the quality of my programming. I was consequently committed to using it for `SCons` development.

Aegis provides a number of things, including:

- A flexible source code control and branching model.
- A defined process with separate development, review and integration steps.
- A distributed development model based on distribution of atomic change sets.

The single most important reason for using Aegis, however, is its management of automated tests as part of the development process.

Testing, Testing, Testing

The `SCons` project has made extensive use of automated tests from day one, taking inspiration mostly from Aegis, partly from the eXtreme Programming model, and with a little home-brew scripting for glue.

Testing Criteria

The underlying criteria for testing changes to the `SCons` code are taken from Aegis:

- Every change must have one or more new or modified tests checked in along with the code.
- The new code being checked in must pass all of the new and/or modified tests.
- The *old*, already checked-in code in must *fail* all of the new and/or modified tests.
- The new code being checked in must pass all unmodified, already checked-in tests.

In practice, these restrictions can be overridden as necessaryfor example, when changing comments or documentation.

The criterion that surprises many people is having the old code fail the tests in the change. This makes sure that the new tests or modified tests really do exercise the bug fix or feature being added by the change.

Together, these criteria ensure that every newly checked-in version `SCons` conforms to defined behavior, as defined by the tests. Whenever a bug is found, its fix is checked in with a new or modified test that guarantees the bug will not recur in the future. We have already built up a regression test base of almost 90 tests that cover the vast majority of `SCons`' functionality.

Testing Infrastructure

Testing standards are no good if they're too much of a burden for developers, who will at best work around or ignore the testing requirements, or at worst stop contributing code and go join a project that's more fun. To this end, good testing infrastructure that makes it easy to write tests is crucial.

`SCons` development uses two development methodologies, one for the individual modules in the build engine, and the other for end-to-end tests of the `SCons` script.

For the build engine modules, we use PyUnit. Every change to a build engine module must have a change to its corresponding unit tests, which live side-by-side in a separate file that imports module. As we build up a large body of unit tests, this ensures that the build engine will perform correctly whenever someone uses it in some application other than the `SCons` script itself.

For end-to-end script tests, we have developed two modules to make writing tests easy. The first, `TestCmd.py`, is a generic module for testing commands or scripts (in any language, not just Python). The second module, `TestScons.py`, is a subclass of the generic `TestCmd.py` module. `TestScons.py` takes care of initialization and displaying error conditions specific to testing `SCons`.

In practice, simple tests only need to initialize a test object, use the object to write some input files, run `SCons`, and then check whatever criteria determine whether the test passed or failed. A complete test of the `Program` method, for example, looks like this:

```
test = TestSCons.TestSCons()

test.write('SConstruct',
"""env = Environment()
env.Program(target = 'foo', source = 'foo.c')
""")

test.write('foo.c',
"""
int
main(int argc, char *argv[])
{
    argv[argc++] = "-"; /* dummy use of args */
    printf("foo.c successfully compiled\\n");
    exit (0);
}
""")

test.run(arguments = 'foo') # runs SCons

test.run(program = test.workpath('foo'))

test.fail_test(test.stdout() != "foo.c successfully compiled\n")

test.pass_test()
```

SourceForge

Registration of the `SCons` project was approved at SourceForge on 29 June 2001. Within a week, the initial code base was checked in, mailing lists were created, and the web site was set up. We started making use of the task-list manager to track what we had to finish for initial release.

The obvious complication was how to use structured testing methodology of Aegis when SourceForge uses CVS for source control. Not using the SourceForge CVS tree would have had two significant disadvantages: one, missing out on the archiving and central location in the event of disaster; two, people coming to the SourceForge project page wouldn't be able to browse the source. The latter was particularly important in the early stages of development, in order to avoid any impression that this was Yet Another Project that starts with a bang and then dwindles as the initial enthusiasm starts to wear off.

The solution was to use the SourceForge CVS repository for read-only access to the source. `SCons` developers are welcome to use CVS for their development, but the changes are *not* committed to the SourceForge repository. Instead, patches are sent to the integrator for processing through Aegis. When the change has been integrated into the Aegis repository, a home-brew script translates the Aegis change into a virtual shell script of commands that copy the necessary files from Aegis and check them in to CVS at SourceForge.

(In practice, write access is not actually disabled for registered developers, but if they do make any changes directly at SourceForge, they can be overwritten at the next Aegis update.)

Future Directions

There are a number of things we would like to do to continue to improve `SCons` in the future.

Distutils Cooperation

There is a certain amount of overlap between what `SCons` does to search out and make use of various compilers on a system, and the impressively complete job that the Distutils do of describing much the same thing. Collaborating to provide some sort of common interface between the two tools would benefit both tools.

Additional Builder Support

Adding additional builders would broaden the potential user base. In rough order of importance:

Java

Given the popularity of Java, support for it would greatly increase the appeal of `SCons` in the large community of Java users.

Good support for Java is, however, a tricky proposition. Because the Java compiler can make decisions about compiling other files based on what classes it finds in a file, it behaves "unpredictably" from the point of view of an outside build tool like `SCons` or `Make`. Some sort of sophisticated scanning of Java source code to identify what other classes are likely to be compiled would be an obvious first step, but notice that here `SCons` would be scanning the file to find additional targets to be built. This is the inverse of the sort of `#include` scanning performed for C files, in which `SCons` is looking for additional *dependencies*.

Documentation toolchains

A number of early adopters are using `SCons` to build documents from TeX or DocBook source files. Built-in support for various documentation toolchains would be an obvious boon for many people.

C#

The reality is that anything that Microsoft does will doubtless have a wide audience. Turning `SCons`' back on that would be cutting off its nose to spite its face.

Fortran

Despite the fact that `SCons` is no longer directly associated with Software Carpentry, it still shares the same goal: to make programming easier for more than just programmers. To that end, good Fortran support would help a great many physical scientists and other computer *users* out there who still rely on Fortran for a great deal of their work.

Database Interface

The Nodes in an `SCons` dependency graph aren't only restricted to files. Creating an interface to mSQL or MySQL databases would allow the possibility of updating external files in response to changes in database fields, or vice versa. This could be handy, for example, for generating a cache of static web pages from a database that only need re-generating when the appropriate database objects change.

Tool Integration

`SCons` should work well with as many popular Integrated Development Environments (IDEs) and tool chains as possible: Komodo, Microsoft Visual Studio, ClearCase, etc. Suggestions for additional tools are welcome.

Makefile Interface

Because the `SCons` Build Engine can be embedded in any Python interface, there isn't any technical reason why a `Makefile` interpreter couldn't be written in Python and use the `SCons` Build Engine for its dependency analysis.

Proof-of-concept for the idea already exists. Gary Holt's `make++` (also known as `makepp`) is a Perl implementation of just such a `Makefile` interpreter. It could possible serve as a model for a Python version, in much the same way the `Cons` design served as the prototype for `SCons`.

Summary

This paper has introduced `SCons`, a next-generation build tool with a modular, embeddable architecture and a direct Python interface. `SCons` has a global view of the dependencies in a source tree, uses MD5 signatures to decide if derived files are out of date, and automatically scans files for dependencies, all of which make `SCons` builds exceptionally reliable. The `SCons` development methodology has been described, notable for its emphasis on automated regression testing to ensure a robust and reliable tool from day one. Several future directions for `SCons` have also been discussed.

Acknowledgements

First, many thanks to the great group of developers who dove in right from the beginning and have contributed the code and ideas to make `SCons` a success: Chad Austin, Charles Crain, Steve Leblanc, and Anthony Roach. Thanks also to those on the scons-devel mailing list who have contributed greatly to the discussion, notably including David Abrahams, Trent Mick, and Steven Shaw.

`SCons` would not exist today without the pioneering work of Bob Sidebotham on the original `Cons` tool, and without Greg Wilson's having started the Software Carpentry contest.

Thanks also to Peter Miller for: Aegis; the testing discipline that it enforces, without which creating a stable but flexible tool would be impossible; the "Recursive Make Considered Harmful" paper which led me to experiment with `Cons` in the first place.

References