JCons Manual

Release 1.3

Johan Holmberg

May 06, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

JCons is a Make replacement, i.e. a program for building other programs. JCons is designed to build C/C++ programs, but any files that are produced in a well defined manner can be created under the control of JCons. As can be guessed from the name, JCons' main inspiration is Cons, the make tool written in Perl (described in the Spring 1998 issue of "The Perl Journal"). To my knowledge Cons was the first make tool to combine a number of interesting ideas:

- The build description files can be *programs* written in an existing language (e.g. Perl), instead of using a description language that is unique to the make tool (e.g. Makefile-syntax or Jam-files).
- Cryptographic checksums can be used to get more reliable dependency checking than that provided by Make, where file timestamps are used to decide if a file should be rebuilt.
- Automatic detection of `#include` file dependencies makes building of C/C++ programs much more reliable.

JCons tries to steal most of the good ideas of Cons. But there are important differences too, for example:

- JCons uses Python as its primary language for description files.
- JCons can build in parallel.
- JCons is much faster than Cons when doing an up-to-date check.

The ideas pioneered by Cons are not so special today over fifteen years later. A number of similar tools exist, most notably SCons (also inspired by Cons). Compared to SCons, JCons is quite minimalistic and lacks many features.

So why develop JCons? First, because it was fun. Writing a make tool seems quite easy in the beginning. But to get a reliable, fast and useful tool requires attention to many small details. This has been an interesting challenge. Second, because I believe JCons does some things better than both Cons and SCons. JCons is for example much faster than both Cons and SCons.

1.1 About this document

The rest of this document tries to describe how to use JCons. First a simple example will be given to illustrate the overall way JCons works. Then a guide to the different JCons features follows. And finally a reference part will describe all the details.

If you have used Cons before, you should recognize many things. The biggest difference from Cons is of course that JCons uses Python as scripting language instead of Perl.

The examples in the text have actually been executed as real commands. This should ensure that they stay correct at all times.

1.2 How JCons works

1. JCons looks for a file `construct.py`, and loads it as Python code. The file has the same role for JCons as the `Makefile` has for Make.
2. The code in `construct.py` should call methods in the `Cons` class. These methods are used to inform JCons what things there are to build.
3. Once the `construct.py` has been read, JCons uses the collected information to start its build engine.

A minimal example of using JCons is given below:

```
# construct.py
e = Cons()
e.command("readme.pdf", "readme.ps",  "ps2pdf %INPUT %OUTPUT")
e.command("readme.ps",  "readme.txt", "enscript -q %INPUT -o %OUTPUT")
```

The calls to `Cons.command()` tells JCons how the output files can be produced from the input files:

```
$ echo a line of text > readme.txt   # create source file
$ jcons
enscript -q readme.txt -o readme.ps
ps2pdf readme.ps readme.pdf
$ jcons                              # nothing to do 2nd time
jcons: up-to-date: .
$ echo one more line >> readme.txt
$ jcons                              # input file has changed
enscript -q readme.txt -o readme.ps
ps2pdf readme.ps readme.pdf
```

All methods of the `Cons` class are of this type. Each method is useful in a different situation: `program` to build a program, `static_library` to create a library, etc. But all methods could in principle have been implemented on top of the `command` method.

CHAPTER

TWO

A TOUR OF JCONS

2.1 Basics

Suppose we have a program consisting of two modules `main.c` and `mod1.c`, and a header file `mod1.h`:

```
/* main.c */
#include <stdio.h>
#include "mod1.h"
int main() {
    printf("%s from main\n", GREETING);
    mod1_greeting();
    return 0;
}

/* mod1.c */
#include <stdio.h>
#include "mod1.h"
void mod1_greeting() {
    printf("%s from mod1\n", GREETING);
}

/* mod1.h */
#define GREETING "hello"
extern void mod1_greeting();
```

To build it with JCons we could create the following file:

```
# construct.py
e = Cons()
e.program("prog", "main.c", "mod1.c")
```

When JCons is invoked, the program `prog` will be built in the following manner:

```
$ jcons
gcc -c main.c -o main.o
gcc -c mod1.c -o mod1.o
gcc -o prog main.o mod1.o
$ jcons
jcons: up-to-date: .
$ ./prog
hello from main
hello from mod1
```

The second time everything is "up to date", so nothing is done. If any of the involved files changes, JCons will detect that an re-build the necessary files:

```
$ echo "void a_change() {}" >> main.c      # change source file
$ jcons
gcc -c main.c -o main.o
```

```
gcc -o prog main.o mod1.o

$ rm main.o                                  # remove object file
$ jcons
gcc -c main.c -o main.o

$ echo GARBAGE > mod1.o                      # destroy content of object file
$ jcons
gcc -c mod1.c -o mod1.o

$ rm prog                                    # remove program
$ jcons
gcc -o prog main.o mod1.o
```

In some of the examples above only an object file is rebuild. Since the newly created object file has the same content as before, no re-linking is needed. For users of Make this might seem odd, but this is one of the features of tools like JCons. JCons tracks changes to the **contents** of the files. The only function of timestamps is as an indicator of change (a file *might* have changed if its timestamp has changed). If we only change the timestamps of files, nothing will be re-built:

```
$ touch main.c
$ jcons
jcons: up-to-date: .

$ touch mod1.o
$ jcons
jcons: up-to-date: .

$ touch prog
$ jcons
jcons: up-to-date: .
```

The module `mod1.c` has a header file `mod1.h` included by the C file itself and by `main.c`. JCons will detect changes to that header file:

```
$ jcons                                      # up-to-date before change
jcons: up-to-date: .

$ perl -i -pe 's/hello/goodbye/' mod1.h      # change header file
$ jcons
gcc -c main.c -o main.o
gcc -c mod1.c -o mod1.o
gcc -o prog main.o mod1.o
$ ./prog
goodbye from main
goodbye from mod1
```

This works because JCons looks for `#include` lines in the source files and automatically adds dependencies for the files it finds. If we would like to build the program in a debug flavor, we need to change the `construct.py` file:

```
# construct.py
e = Cons(CFLAGS = "-DDEBUG -g", LINKFLAGS = "-g")
e.program("prog", "main.c", "mod1.c")

$ jcons
gcc -DDEBUG -g -c main.c -o main.o
gcc -DDEBUG -g -c mod1.c -o mod1.o
gcc -o prog -g main.o mod1.o
```

As can be seen, the configuration variable `CFLAGS` affects how a C program is built. Variables like `CFLAGS` can also be set temporarily on the command line:

```
$ jcons CFLAGS="-O2"                    # command line override
gcc -O2 -c main.c -o main.o
gcc -O2 -c mod1.c -o mod1.o
gcc -o prog -g main.o mod1.o

$ jcons CFLAGS="-Wall"                  # another override
gcc -Wall -c main.c -o main.o
gcc -Wall -c mod1.c -o mod1.o
gcc -o prog -g main.o mod1.o

$ jcons CFLAGS="-Wall"                  # same setting again
jcons: up-to-date: .

$ jcons                                 # back to normal
gcc -DDEBUG -g -c main.c -o main.o
gcc -DDEBUG -g -c mod1.c -o mod1.o
gcc -o prog -g main.o mod1.o
```

As can be seen in the examples, a change in command line triggers a rebuild. JCons includes the command line in the dependency calculation. JCons knows which files are "generated files" and can remove them if asked to do that (with the `-r` option):

```
$ jcons -r                              # remove generated files
Removed main.o
Removed mod1.o
Removed prog
```

It is also possible to force a full rebuild:

```
$ jcons                                 # normal build
gcc -DDEBUG -g -c main.c -o main.o
gcc -DDEBUG -g -c mod1.c -o mod1.o
gcc -o prog -g main.o mod1.o

$ jcons                                 # nothing to do
jcons: up-to-date: .

$ jcons  --always-make                  # forced build
gcc -DDEBUG -g -c main.c -o main.o
gcc -DDEBUG -g -c mod1.c -o mod1.o
gcc -o prog -g main.o mod1.o
```

2.2 Several output files

A command can have several output files. The method `Cons.command()` can handle this situation:

```
e = Cons()
e.command(["parse.tab.c", "parse.tab.h"], "parse.y", "bison -d parse.y")
```

Here the output argument is an array of files produced by the command. If any of those files doesn't exit or is out-of-date the command has to be run. Both the input- and output-argument to `Cons.command()` can be arrays instead of single values. JCons understands how to deal with those cases too.

2.3 C++ programs

If the some of the source files had been C++ rather than C files, the `construct.py` would look almost the same:

```
# construct.py
e = Cons()
e.program("prog", "main.cpp", "mod1.c")
```

When running JCons we then get:

```
$ jcons
g++ -c main.cpp -o main.o
gcc -c mod1.c -o mod1.o
g++ -o prog main.o mod1.o
```

Note that `g++` is used to compile the C++ file, and also used when linking.

2.4 Include files

The include path to a C/C++ compiler is typically given by `-I` options on the command line. For JCons to be able to calculate the #include dependencies, the directories have to be specified via a variable `CPPPATH`:

```
# construct.py
e = Cons(CPPPATH = ["dir1", "dir2"])
e.program("prog", "main.c")
```

When JCons is executed the values in `CPPPATH` are translated into `-I ''` options on the command line. Suppose the file ``main.c`` looks like:

```
/* main.c */
#include <main.h>
int main() { return EXIT_CODE; }
```

Running JCons we get:

```
$ mkdir -p dir1 dir2
$ echo "#define EXIT_CODE 2" > dir2/main.h
$ jcons
gcc -I dir1 -I dir2 -c main.c -o main.o
gcc -o prog main.o

$ echo "#define EXIT_CODE 1" > dir1/main.h
$ jcons
gcc -I dir1 -I dir2 -c main.c -o main.o
gcc -o prog main.o
```

The second time JCons realizes that the `main.h` located in `dir1` is going to be used, and recompiles `main.c`.

2.5 Libraries

TODO: write this section ....

2.6 Variant builds

Often a program should be built in several "flavours", e.g. a debug and and a release version. Then the object files and executables need to be stored in different places for each flavour. JCons makes it easy to change where the output is placed in several ways:

1. in a separate build directory tree (by using `BUILD_TOP`)
2. in separate sub-directories (by using `BUILD_SUBDIR`)

3. with different filename suffixes (by using `BUILD_SUFFIX`)

If `BUILD_TOP` is used, we get:

```
# construct.py
e = Cons(BUILD_TOP = "build/release")
e.program("prog", "main.c", "lib/mod1.c")

$ jcons
gcc -c lib/mod1.c -o build/release/lib/mod1.o
gcc -c main.c -o build/release/main.o
gcc -o build/release/prog build/release/main.o build/release/lib/mod1.o
```

To build both a release and a debug version, we can use normal Python scripting:

```
# construct.py
flavors = [
    ["release", "-O2 -DNDEBUG"],
    ["debug",   "-DDEBUG"],
]
for flavor, cflags in flavors:
    e = Cons(BUILD_TOP = "build/" + flavor, CFLAGS = cflags)
    e.program("prog", "main.c", "lib/mod1.c")
```

With this file the build would look like:

```
$ jcons --always-make
gcc -DDEBUG -c lib/mod1.c -o build/debug/lib/mod1.o
gcc -DDEBUG -c main.c -o build/debug/main.o
gcc -o build/debug/prog build/debug/main.o build/debug/lib/mod1.o
gcc -O2 -DNDEBUG -c lib/mod1.c -o build/release/lib/mod1.o
gcc -O2 -DNDEBUG -c main.c -o build/release/main.o
gcc -o build/release/prog build/release/main.o build/release/lib/mod1.o
```

If `BUILD_SUBDIR` was used instead of `BUILD_TOP`, we would get:

```
# construct.py
e = Cons(BUILD_SUBDIR = "release")
e.program("prog", "main.c", "lib/mod1.c")

$ jcons
gcc -c lib/mod1.c -o lib/release/mod1.o
gcc -c main.c -o release/main.o
gcc -o release/prog release/main.o lib/release/mod1.o
```

or if `BUILD_SUFFIX` was used:

```
# construct.py
e = Cons(BUILD_SUFFIX = "release")
e.program("prog", "main.c", "lib/mod1.c")

$ jcons
gcc -c lib/mod1.c -o lib/mod1-release.o
gcc -c main.c -o main-release.o
gcc -o prog-release main-release.o lib/mod1-release.o
```

Note that only methods producing object files or executables (e.g. `Cons.program()` or `Cons.static_library()`), are affected by the `BUILD_*` variables. `Cons.command()` does not look at those variables.

2.7 Cache of build results

JCons can cache the generated files in a special directory. If the same file is about to built again later, JCons can replace the actual command with a copy operation from the cache directory. This is much faster, and avoids needlessly re-executing the same command with the same input several times.

TODO: add example

2.8 The construct.py file

JCons reads a file `construct.py` (or another file specified with an `-f` option). This file is a normal Python file where methods of the `Cons` class can be called. The purpose of the file is to tell JCons what there is to build (i.e. help build the directed acyclic graph (DAG) describing the dependencies). It is of course possible (but pointless) to do something entirely different in the script, but then JCons would not know what to do:

```
# construct.py
print("hello world")          # pointless use of JCons
exit(0)

$ jcons
hello world
```

The first thing to do in a `construct.py` file is to create an object of the `Cons` class. Then methods can be called on that object to tell JCons what there is to build. There are different methods for different needs (e.g. `Cons.command()`, `Cons.object()`, `Cons.objects()`, `Cons.static_library()`, `Cons.program()`). The following example will demonstrate:

```
e = Cons()

# C program with two modules
e.program("foo", "foo.c", "bar.c")

# generic command
e.command("bar.pdf", "bar.ps", "ps2pdf %INPUT %OUTPUT")

# a library
e.static_library("mylib", "x.cpp", "y.cpp", "z.cpp")
```

The constructor of the `Cons` class can take optional named arguments. For example:

```
e1 = Cons(CFLAGS = "-g -Wall")
e1.program("foo", "foo1.cpp", "foo2.c")

e2 = Cons(CFLAGS = "-O2 -Wall", CXXFLAGS = "-O2")
e2.program("bar", "bar1.cpp", "bar2.c")
```

2.9 conscript.py files

A larger application will be spread over a number of directories. Each directory may produce a program or a library used by some program. To handle this situation, the main `construct.py` file can include other subsidiary files (typically called `conscript.py` in the tradition from Cons):

```
# construct.py
e = Cons()
e.program("prog", "main.c", "util/util.a")
Cons.include("util/conscript.py")
```

```
# util/conscript.py
e = Cons()
e.static_library("util", "util.c")
```

With these files we get:

```
$ jcons
gcc -c main.c -o main.o
gcc -c util/util.c -o util/util.o
ar rc util/util.a util/util.o
gcc -o prog main.o util/util.a
```

JCons maintains *one* global DAG of all dependencies. All included `conscript.py` file contribute to this dependency graph.

CHAPTER

THREE

REFERENCE

3.1 Handling of PATH & ENV

JCons does not alter the environment in any way. So the environment an executed command sees, is the same as the one in effect when `jcons` was started. And JCons does not find compilers and other tools in some magic way. It is the responsibility of the JCons user to have a suitable PATH before invoking JCons. It is of course possible to set the PATH at the start of the `construct.py` too, the normal Python way:

```
# construct.py
import os
os.environ["PATH"] += ":/some/path/bin"
e = Cons()
e.program("prog", "main.c", "mod1.c")
```

3.2 Evaluation of %VAR variables

When a Cons object is created, it is also given a number of variable settings that affects how things are built.

TODO: write more about evaluation, predefined variables, ...

3.3 Flattening of parameters

Several of the methods of a Cons object expect a list of files (e.g. `Cons.program()`). The list of files can be "nested". JCons will automatically "flatten" the list. The following examples are all equivalent:

```
e.program("prog", "f1.c", "f2.c", "f3.c", "f4.c", "f5.c")

e.program("prog", ["f1.c", "f2.c", "f3.c", "f4.c", "f5.c"])

f12 = ["f1.c", "f2.c"]
f45 = ["f4.c", ["f5.c"]]
e.program("prog", f12, ["f3.c"], f45)
```

JCons flattens file parameters nested as deep as five levels. This is an arbitrary limit, just to avoid "recursive" lists.

3.4 Nested Cons parameters

The way a file is compiled is affected by the Cons object used when calling a method (e.g. `Cons.program()`). Sometimes a program may consist of different groups of files that should be compiled differently. Suppose for example that your application consists of two sets of files that should be built in different ways. JCons handles this by allowing nested Cons parameters:

```
e = Cons()
e_foo = Cons(CFLAGS = "-DFOO")
e_bar = Cons(CFLAGS = "-DBAR")

foo_srcs = ["foo1.c", "foo2.c", ... "foo30.c"]
bar_srcs = ["bar1.c", "bar2.c", ... "bar40.c"]

e.program("prog", [e_foo, foo_srcs], [e_bar, bar_srcs], "mylib.a")
```

Here the files `foo*.c` will be built using the `-DFOO` setting, and the files `bar*.c` will be built using the `-DBAR` setting. The link step will use the first created Cons object (the variable `e`). For each file, the "closest enclosing" and preceding Cons object will be used.

3.5 Cons methods

class Cons (***kwargs*)

This is the constructor, creating an object of type `Cons`. Most other methods described below are **instance methods** on the objects returned by this constructor. The constructor takes an optional hash-argument with settings of configuration variables affecting how things are to be built.

A `Cons` object is meant to capture a way of building things, e.g. a debug- or release-build, or the use of a specific compiler. You are free to create as many `Cons` objects as you need. An example demonstrates some typical uses:

```
e1 = Cons()
e1.program("foo1", "foo1.cpp", "bar1.c")

e2 = Cons(CFLAGS = "-g", CXXFLAGS = "-g")
e2.program("foo2", "foo2.cpp", "bar2.c")
```

`Cons`**.clone** (***kwargs*)

This method creates a copy of an existing object, but modified by the settings given as arguments. A typical use is to make a slight modification of an already exiting object:

```
e1 = Cons(A="1", B="2", C="3")
e2 = e1.clone(B="22", D="4")
```

This is almost the same as the following:

```
e1 = Cons(A="1", B="2", C="3")
e2 = Cons(A="1", B="22", C="3", D="4")
```

`Cons`**.command** (*target*, *source*, *command*)

This is the most basic way of describing how a "target" is built from a "source". The command needed to build the target is specified explicitly:

```
e = Cons()
e.command("bar.pdf", "bar.ps", "ps2pdf bar.ps bar.pdf")
```

Instead of specifying the filenames in two places, the symbols `%INPUT` and `%OUTPUT` can be used in the command. JCons will replace these symbols automatically with the actual filenames before executing the command:

```
e = Cons()
e.command("bar.pdf", "bar.ps", "ps2pdf %INPUT %OUTPUT")
```

Both the target and the source parameters may be an array of files. So a command taking two input files, and producing three output files can be handled:

```
e = Cons()
e.command(["out1.txt", "out2.txt", "out3.txt"],
```

```
        ["in1.txt", "in2.txt"],
        "some_program  ...some_parameters...")
```

The `%INPUT` and `%OUTPUT` variables can be used here too, they will be set to a space separated list of the input/output files.

TODO: describe use of ":uses_cpp".

`Cons.`**`program`**(*target*, *source1*, ... *sourceN*)

TODO: make this section more "reference style"

A common scenario is that a C/C++ program should be built from sources. JCons has a special method named `program()` for this. The same effect can in principle be accomplished with a number of calls to the `command()` method but it would be more clumsy. The most basic `program()` usage looks like:

```
e = Cons()
e.program("prog", "main.c", "mod1.c")
```

This will build an executable `prog` from the two source files `main.c` and `mod1.c`. The compiler settings used are the ones given by the `Cons` object used. In the example above with no parameters to the constructor, the "default" compiler for the platform will be chosen. On Mac OS X it might look like:

```
$ jcons .
gcc -c main.c -o main.o
gcc -c mod1.c -o mod1.o
gcc -o prog main.o mod1.o
```

The sources can be specified as individual arguments to the `program` method, or as an array. If the list of sources is long it might be convenient to use an array:

```
srcs = ["foo1.cpp", "foo2.cpp", ... "foo99.cpp"]
e.program("prog", srcs)
```

The "sources" can also be object files or library files:

```
e.program("prog", "foo.cpp", "bar.o", "mylib.a")
```

This method returns the "target", i.e. the program built. This is almost the same as the target parameter, but is affected by the `BUILD_*` symbols, and on Windows, an ".exe" file suffix has been added automatically.

This method is affected by the setting of `BUILD_TOP`, `BUILD_SUBDIR` and `BUILD_SUFFIX` (see the *Variant builds* section).

`Cons.`**`object`**(*target*, *source*)

To build an object file from a source file with explicit control of the object filename, the `object()` method can be used:

```
e1 = Cons()
e1.object("foo.o", "foo.c")

e2 = Cons(CFLAGS = "-g")
e2.object("foo-debug.o", "foo.c")
```

The method does not return anything useful.

`Cons.`**`objects`**(*source1*, *source2*, ... *sourceN*)

The `objects()` method tells JCons that a number of source files should be built, and returns a list of the object files produced:

```
e = Cons()
objs = e.objects("foo.c", "bar.c", "frotz.c")
e.program("foo", objs)
```

The lines above have the same effect as:

```
e = Cons()
objs = e.program("foo", "foo.c", "bar.c", "frotz.c")
```

This method is affected by the setting of BUILD_TOP, BUILD_SUBDIR and BUILD_SUFFIX (see the *Variant builds* section).

Cons.**static_library**(*library*, *source1*, *source2*, ... *sourceN*)
The static_library method tells JCons to build a static library from a number of source or object files:

```
e = Cons()
e.static_library("libfoo", "foo.c", "bar.c", "frotz.c")
```

TODO: describe ".a" handling TODO: describe "lib*" handling TODO: example using the library

This method is affected by the setting of BUILD_TOP, BUILD_SUBDIR and BUILD_SUFFIX (see the *Variant builds* section).

Cons.**depends**(*target*, *source*)
Tell JCons that "target" depends on "source", even if JCons can't find this out by itself. This method is only useful as a complement to another method call, e.g. command or program. depends by itself has no way of of telling which command should be executed if the dependency "fires".

TODO: example

Cons.**exe_depends**(*target*, *source*)
Tell JCons that the program "target" depends on "source" *when executed*. If another build rule uses "target" as the command to execute, it will need to be rerun if the "source" has changed.

Cons.**install**(*target*, *source*)
Copy the "source" to "target".

Cons.**install**(*target_dir*, *source*)
Copy the "source" to "target_dir".

Cons.**include**(*filename*)
Read a conscript.py file in a sub-directory.

3.6 Configuration Variables

JCons "knows" about a number of variables. These are listed here. Variables beginning with an "_" are set by JCons too.

AR The name of the ar(1) command to use when building static libraries. (default: "ar")

AR_CMD The full command used to run ar(1). Uses AR and AR_FLAGS.

AR_FLAGS The options used when running ar(1). (default: "rc")

CC The name of the C compiler to use. (default: "gcc")

CC_CMD The full command used to run the C compiler. Uses CC and CFLAGS.

CC_LINK bla bla

CFLAGS bla bla

CXX bla bla

CXXFLAGS bla bla

CXX_CMD bla bla

CXX_LINK bla bla

EXE_EXT bla bla

INPUT bla bla

LIB_EXT bla bla

OBJ_EXT bla bla

OUTPUT bla bla

_CPP_INC_OPTS Set by JCons from the `CPPPATH` variable.

CHAPTER

FOUR

BACKGROUND

4.1 Why Make?

Why are Make-like programs needed at all? The first and most obvious answer is: *to save time*. A small application can easily be built by a script, performing all steps every time it is invoked:

```
#!/bin/sh -e
g++ file1.cpp -o file1.o
g++ file2.cpp -o file2.o
g++ file3.cpp -o file3.o
g++ -o prog file1.o file2.o file3.o
```

Running this script may perhaps take a couple of seconds. The fact that we re-compile all files even if just one file has changed is not a problem. And if we are not sure that `prog` is up-to-date, we can run the script once again just to be sure to get an up-to-date `prog`. But what if we have a project with 800 source files? In principle we could build this program too with a simple script. But now the sheer number of files makes this take much longer (perhaps as long as an hour). We can no longer run the script just to be sure the program is up-to-date. And waiting an hour after just changing one source file is not a real option. A Make-like program on the other hand would detect that just one file had changed, and would re-compile that single file and then re-link the application, probably in less than a minute (instead of an hour). This is a huge time saving.

Another need for a Make-like program is: *to make sure that all files are built in the right order*. In a large project it may not be obvious exactly what order of commands is needed to produce the final program. Some source files may for example be generated by other tools, which in turn may be built as part of the whole build process. Keeping track of all dependencies "by hand" is difficult, and there is an definite risk that the final program will be built incorrectly.

Make-like programs solve the problems described above by specifying **declaratively** how different target files depend on their source files. It is then up to the Make-like program to decide which commands should be executed and in what order. The dependencies form a DAG (a directed acyclic graph), and there are well-known algorithms to traverse such a graph in the right order.

4.2 Why not Make?

So if Make solves the build problem, and essentially does it "the right way", why would there be a need for other tools than Make?

First: *lack of global view of all dependencies*. A large software project will span a number of different directories. The traditional Make-solution is to have a `Makefile` in each subdirectory and have the top `Makefile` orchestrate the build process by recursively calling Make in each subdirectory. Each Make-instance will then only have a partial view of the total dependency tree. This can easily lead to situations where the programmer feels compelled to invoke Make several times, just to make sure everything is updated correctly, in case the Makefiles don't catch all global dependencies correctly (see Peter Millers paper: "...").

Second: *Make-syntax is a poor "programming language"*. Originally Make had a simple declarative syntax. But as time went by, the need for more "power" have lead to addition of a number of new features. This can be seen

for example in GNU Make, the de facto standard in open source projects. GNU Make has got many programming language-like features. This "make on steroids" is very powerful, but the Makefiles often looks awful.

Third: *poor file dependency tracking*. A basic assumption in Make is that comparing file timestamps is a good way to see if a file need to be rebuilt. It is easy to "fool" Make that a file is up-to-date (just "touch" the file). Or if a source file is accidentally re-written to disk without any actual changes, a whole project may need to be rebuilt because of the changed timestamp.

Fourth: *changed command line is not taken into account when deciding if a target file need to be updated.* After changing some compiler option in a Makefile, a full rebuild may be needed to make sure all affected object files are updated correctly.

Fifth: *"#include" dependencies are not tracked by Make*. Sure, there are ways to compensate for this by having artificial entries in the Makefile, calling the compiler asking it for these dependencies. But this needs a lot of boilerplate code in the Makefile.

Sixth: implicit rules are bad ...

For some of the defects in Make there are workarounds today (e.g. `#include` file tracking). Others could in principle be solved in Make (e.g. using cryptographic checksums instead of timestamps). But the first two are not easily fixed within Make: 1) the poor programing language, and 2) the lack of global dependency view.

CHAPTER

FIVE

APPENDIX

5.1 Speed benchmarking

One example: on a program consisting of around 800 C++ files I have measured the time to verify that the program is "up-to-date". I got the following numbers: Cons 42 s, SCons 102 s, JCons 1.03 s. These numbers were measured on an IMac G5 2.0 GHz from 2005 running MAC OS Leopard. In all three cases the build descriptions just use the most basic methods available: `Cons.program()`, `Cons.static_library()` and `Cons.objects()` (named slightly different in the different tools).

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

INDEX