The Waf Book (v1.5.19)

Contents

Introduction

A word on build systems

As software is becoming increasingly complex, the process of creating software is becoming more complex too. Today's software uses various languages, requires various compilers, and the input data is spread into many files.

Software is now used to express the process of building software, it can take the form of simple scripts (shell scripts, makefiles), or compilers (CMake, Qmake), or complete applications (SCons, Maven, Waf). The term 'build system' is used to design the tools used to build applications.

The Waf framework

Build systems make assumptions on software it is trying to build, and are typically limited where it comes to processing other languages or different projects. For example, Ant is better suited than Make for managing Java projects, but is more limited than Make for managing simple c projects. The programming tools are evolving constantly, making the creation of a complete build system for end-users impossible.

The Waf framework is somewhat different than traditional build systems in the sense that the focus is to support indirectly the major usecases encountered when working on a software project, without supporting a language directly. Waf is essentially a library of components that are suitable for use in a build system, with an emphasis on extensibility. Although the default distribution contains various plugins for different programming languages and different tools (c, d, ocaml, java, etc), it is by no means a frozen product. Creating new extensions is both a standard and a recommended practice.

Objectives of this book

The objectives of this book is to expose the use of the Waf build system though the use of Waf in practice, the description of the Waf extension system, and an overview of the Waf internals. We hope that this book will serve as a reference for both new and advanced users. Although this book does not deal with build systems in general, a secondary objective is to illustrate quite a few new techniques and patterns through numerous examples.

The chapters are ordered by difficulty, starting from the basic use of Waf and Python, and diving gradually into the most difficult topics. It is therefore recommended to read the chapters in order. It is also possible to start by looking at the examples from the waf distribution before starting the reading.

Chapter 1

Getting Started

1.1 Obtaining Waf

The Waf project is located on Google Code. The current Waf version requires an interpreter for the Python programming language such as cPython 2.3 to 3.1 or Jython >= 2.5. Note that Waf requires bzip2 compression support, which may be unavailable in self-compiled cPython installations.

1.1.1 Using the Waf binary

The Waf binary does not require any installation whatsoever. It may be executed directly from a writeable folder. Just rename it into ’waf’ if necessary:

```
$ wget http://waf.googlecode.com/files/waf-1.5.19
$ mv waf-1.5.19 waf && chmod 755 waf
```

1.1.2 Generating the Waf binary from the source code

The Waf binary creation requires a Python interpreter in version 2.4 to 2.6. The source code is processed to support Python 2.3 and Python 3.

```
$ wget http://waf.googlecode.com/files/waf-1.5.19.tar.bz2
$ tar xjvf waf-1.5.19.tar.bz2
$ cd waf-1.5.19
$ ./waf-light --make-waf
```

1.2 Running Waf

The *waf* script can be used directly from a writeable folder:

```
$ ./waf
```

Or, if the executable permissions are not set properly:

```
$ python waf
```

The *waf* file has its own library compressed in a binary stream in the same file. Upon execution, the library is uncompressed in the current directory, in a hidden folder that will be re-created if removed. The naming enables different Waf versions to be executed from the same folders:

```
$ ls -ld .waf*
.waf-1.5.19-2c924e3f453eb715218b9cc852291170
```

By default, the recommended Python interpreter is cPython, for which the supported versions are 2.3 to 3.1. For maximum convenience for the user, a copy of the Jython interpreter in the version 2.5 may be redistributed along with a copy of the Waf executable.

Note
A project containing *waf*, *jython2.5.jar* and the source code may be used almost anywhere.

Warning
The *waf* script must reside in a writable folder to unpack its cache files.

1.3 Installing Waf on a system

Installing Waf on a system is unnecessary and discouraged:

1. Interpreter version: unlike the *waf* file, the installed version will not run on Python 3
2. Operating systems: Waf cannot be installed on Windows (yet)
3. Installation: installation is cumbersome, and requires administrative privileges
4. Waf versions: users rarely install the appropriate software version (too old, too new, bugs)
5. Bugs: projects may depend on unspecified behaviour in particular Waf versions
6. Size: the *waf* file is small enough to be redistributed (about 90kB)

Installation require a Python interpreter in the versions 2.4 to 2.6. It may require administrator or root privileges, depending on the target folder (prefix).

```
# ./waf-light configure --prefix=/usr/
# ./waf-light install
```

Likewise, Waf can be uninstalled using:

```
# ./waf-light --prefix=/usr/
# ./waf-light uninstall
```

Chapter 2

The scripting system

This chapter provides a practical overview of the waf usage in a software project.

2.1 Setting up a project

The Waf projects are structured based on the following concepts:

1. Source directory: directory containing the source files that will be packaged and redistributed to other developers or to end users
2. Build directory: directory containing the files generated by the project (configuration sets, build files, logs, etc)
3. System files: files and folders which do not belong to the project (operating system files, etc)

When Waf is launched, it looks for the top-level project file, which are Python scripts. A valid Waf project requires at least a top-level Waf script file (named *wscript* without any extension) containing the three following elements:

1. top: string representing the project directory. In general, top is set to '.', except for some proprietary projects where the wscript cannot be added to the top-level, top may be set to '../..' or even some other folder such as */checkout/perforce/project*
2. out: string representing the build directory. In general, it is set to *build*, except for some proprietary projects where the build directory may be set to an absolute path such as */tmp/build*. It is important to be able to remove the build directory safely, so it should never be given as '.' or '..'.
3. configure: function called for setting up a project (also known as a *Waf command*).

The Waf script will always look for one valid top-level project file first before considering other project files.

Now let us create a new Waf project in the folder */tmp/smallproject*. The first step is to write a wscript file in */tmp/smallproject/wscript* with the following contents:

```
top = '.'
out = 'build_directory'

def configure(ctx):
        print('→ configuring the project')
```

To use a Waf project for the first time, it is necessary to initialize it. Waf will then validate the project file, and create the cache files for later use (lock file, build directory, store the project options):

```
$ cd /tmp/smallproject ❶
$ tree
.
`-- wscript

$ waf configure ❷
→ configuring the project
'configure' finished successfully (0.021s)

$ tree
.
|-- build_directory/ ❸
|   |-- c4che/ ❹
|   |   |-- build.config.py ❺
|   |   `-- default.cache.py ❻
|   |-- config.log ❼
|   `-- default/ ❽
|--.lock-wscript ❾
`-- wscript
```

❶ To configure the project, go to the folder containing the top-level project file

❷ The execution is called by calling `waf configure`

❸ The build directory was created

❹ The configuration data is stored in the folder *c4che/*

❺ The command-line options and environment variables in use are stored in this file

❻ The user configuration set is stored in this file

❼ Configuration log (duplicate of the output generated during the configuration)

❽ Directory for the generated files (none so far)

❾ Lock file pointing at the relevant project file and build directory

2.2 Adding project commands

In the last section, we have seen the use of the command *configure* to initialize a project. Waf commands are special functions declared in the top-level project file and which may be called explicitly by `waf commandname`. Let us create two new commands *print_ping* and *print_pong* in the project created previously (*/tmp/smallproject/*)

```
top = '.'
out = 'build_directory'

def configure(ctx):
        print('→ configuring the project')

def print_ping(ctx):
        print(' ping!')

def print_pong(ctx):
        print(' pong!')
```

Waf commands always take a single parameter called the *context*, which is used to ease data sharing between the scripts. To execute the new commands:

```
$ cd /tmp/smallproject

$ waf configure
→ configuring the project
'configure' finished successfully (0.001s)

$ waf print_ping
 ping!
'print_ping' finished successfully (0.000s)

$ waf print_pong
 pong!
'print_pong' finished successfully (0.000s)
```

Waf commands may also be executed at once by chaining them:

```
$ waf print_ping print_pong print_ping
 ping!
'print_ping' finished successfully (0.000s)
 pong!
'print_pong' finished successfully (0.000s)
 ping!
'print_ping' finished successfully (0.000s)
```

Waf commands may be repeated several times too:

```
$ waf print_ping print_ping print_ping
 ping!
'print_ping' finished successfully (0.000s)
 pong!
'print_ping' finished successfully (0.000s)
 ping!
'print_ping' finished successfully (0.000s)
```

2.3 Cleaning up a project

Waf itself comes with a predefined command called *distclean* which removes the build directory and the lock file. After calling cleaning a project, it is necessary to configure it once again.

```
$ waf configure
→ configuring the project
'configure' finished successfully (0.001s)

$ waf print_ping
 ping!
'print_ping' finished successfully (0.000s)

$ waf distclean
'distclean' finished successfully (0.001s)

$ waf print_ping
Project not configured (run 'waf configure' first)
```

It is possible to override the behaviour of *distclean* by redefining it in the wscript file. For example, the following will cause it to avoid removing the build files.

```
top = '.'
out = 'build_directory'
```

```
def configure(ctx):
        print('→ configuring the project')

def distclean(ctx):
        print(' Not cleaning anything!')
```

Upon execution:

```
$ waf distclean
 not cleaning anything!
'distclean' finished successfully (0.000s)
```

2.4 Packaging the project sources

The command *dist* is another predefined utility which is used to create an archive of the project. By using the script presented previously:

```
top = '.'
out = 'build_directory'

def configure(ctx):
        print('→ configuring the project')
```

Execute the command *dist* to get:

```
$ waf configure
→ configuring the project
'configure' finished successfully (0.001s)

$ waf dist
New archive created: noname-1.0.tar.bz2 (sha='c16c97a51b39c7e5bee35bb6d932a12e2952f2f8')
'dist' finished successfully (0.091s)
```

By default, the project name and version are set to *noname* and *1.0*. To change them, it is necessary to provide two additional variables in the top-level project file:

```
APPNAME='webe'
VERSION='2.0'

top = '.'
out = 'build_directory'

def configure(ctx):
        print('→ configuring the project')
```

Because the project was configured once, it is not necessary to configure it once again:

```
$ waf dist
New archive created: webe-2.0.tar.bz2 (sha='7ccc338e2ff99b46d97e5301793824e5941dd2be')
'dist' finished successfully (0.006s)
```

The default compression format is bzip2. It may be changed to gzip by using the symbol *gz*:

```
import Scripting
Scripting.g_gz = 'gz'
```

Or *zip* for zip files:

```
import Scripting
Scripting.g_gz = 'zip'
```

2.5 Splitting a project into several files

Although a Waf project must contain a top-level wscript file, the contents may be split into several sub-project files. We will now illustrate this concept on a small project:

```
.
|-- src
|   `-- wscript
`-- wscript
```

The commands in the top-level wscript will call the same commands from a subproject wscript file by calling a context method named *recurse*.

```
top = '.'
out = 'build_directory'

def configure(ctx):
        print('→ configure from the top-level')
        ctx.recurse('src')

def print_ping(ctx):
        print('→ ping from the top-level')
        ctx.recurse('src')
```

Since the folder *src* is not a redistributable project, it is not necessary to duplicate the variables *top* and *out*.

```
def configure(ctx):
        print('→ configure from src')

def print_ping(ctx):
        print('→ ping from src')
```

Upon execution, the results will be:

```
$ cd /tmp/smallproject

$ waf configure print_ping
→ configure from the top-level
→ configure from src
'configure' finished successfully (0.080s)
→ ping from the top-level
→ ping from src
'print_ping' finished successfully (0.009s)
```

2.6 Building, cleaning, installing and uninstalling a project

The *build* command is used for building the actual software. Starting again from the project file */tmp/smallfolder/wscript*:

```
top = '.'
out = 'build_directory'

def configure(ctx):
        print('→ configure from the top-level')

def build(ctx):
        print('building the software')
```

Without surprise, the execution output will look like the following:

```
$ cd /tmp/smallproject

$ waf
Project not configured (run 'waf configure' first)

$ waf configure
→ configure from the top-level
'configure' finished successfully (0.001s)

$ waf build
Waf: Entering directory `/tmp/smallproject/build_directory'
building the software
Waf: Leaving directory `/tmp/smallproject/build_directory'
'build' finished successfully (0.004s)
```

Since the command `waf build` is executed very often, a shortcut is provided to call it implicitly:

```
$ waf
Waf: Entering directory `/tmp/smallproject/build_directory'
building the software
Waf: Leaving directory `/tmp/smallproject/build_directory'
```

The Waf commands `build`, `clean`, `install`, `uninstall` are shortcuts for calling `waf build` with different internal options. They all require the presence of the *build* function in the scripts.

```
$ waf build install uninstall clean
Waf: Entering directory `/tmp/smallproject/build_directory'
building the software
Waf: Leaving directory `/tmp/smallproject/build_directory'
'build' finished successfully (0.004s)
Waf: Entering directory `/tmp/smallproject/build_directory'
building the software
Waf: Leaving directory `/tmp/smallproject/build_directory'
'install' finished successfully (0.003s)
Waf: Entering directory `/tmp/smallproject/build_directory'
building the software
Waf: Leaving directory `/tmp/smallproject/build_directory'
'uninstall' finished successfully (0.002s)
building the software
'clean' finished successfully (0.002s)
```

The meaning of the commands is the following:

1. `build:` process the source code to create the object files
2. `clean:` remove the object files that were created during a build (unlike distclean, do not remove the configuration)
3. `install:` check that all object files have been generated and copy them on the system (programs, libraries, data files, etc)
4. `uninstall:` undo the installation, remove the object files from the system without touching the ones in the build directory

Object file creation and installation will be detailed in the next chapters.

2.7 Customizing the command-line options

The Waf script provides various default command-line options, which may be consulted by executing `waf --help`:

```
$ waf --help
waf [command] [options]

Main commands (example: ./waf build -j4)
  build    : builds the project
  clean    : removes the build files
  configure: configures the project
  dist     : makes a tarball for redistributing the sources
  distcheck: checks if the sources compile (tarball from 'dist')
  distclean: removes the build directory
  install  : installs the build files
  uninstall: removes the installed files

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -j JOBS, --jobs=JOBS  amount of parallel jobs (2)
  -k, --keep            keep running happily on independent task groups
  -v, --verbose         verbosity level -v -vv or -vvv [default: 0]
  --nocache             ignore the WAFCACHE (if set)
  --zones=ZONES         debugging zones (task_gen, deps, tasks, etc)
  -p, --progress        -p: progress bar; -pp: ide output
  --targets=COMPILE_TARGETS
                        build given task generators, e.g. "target1,target2"

  configuration options:
    --prefix=PREFIX     installation prefix (configuration) [default: '/usr/local/']

  installation options:
    --destdir=DESTDIR   installation root [default: '']
    -f, --force         force file installation
```

Accessing a command-line option is possible from any command. Here is how to access the value *prefix*:

```
top = '.'
out = 'build_directory'

def configure(ctx):

        import Options
        print('→ prefix is ' + Options.options.prefix)
```

Upon execution, the following will be observed:

```
$ waf configure
→ prefix is /usr/local/
'configure' finished successfully (0.001s)
```

To define project command-line options, a special command named *set_options* may be defined in user scripts. This command will be called once before any other command executes.

```
top = '.'
out = 'build_directory'

def set_options(ctx):
        ctx.add_option('--foo', action='store', default=False, help='Silly test')

def configure(ctx):

        import Options
        print('→ the value of foo is %r' % Options.options.foo)
```

Upon execution, the following will be observed:

```
$ waf configure --foo=test
→ the value of foo is 'test'
'configure' finished successfully (0.001s)
```

The command context for set_options is a shortcut to access the optparse functionality. For more information on the optparse module, consult the Python documentation

Chapter 3

The configuration system

The configuration step is used to check if the requiremements for working on a project are met. The parameters are then stored for further use.

3.1 Storing configuration sets

Configuration sets are instances of the class *Environment*. That class acts as a wrapper around Python dicts to handle serialization (in human-editable files) and copy-on-write efficiently. Instances of that class are used extensively within waf, and the instances are usually named *env*. Here is how the default configuration set called *env* is usually stored:

```
top = '.'
out = 'build'

def configure(conf):
        conf.env['CXXFLAGS'] = ['-O2'] # key-based access
        conf.env.store('test.txt')
        conf.env.TEST        = 'test'  # attribute-based access

        new = conf.env.__class__()
        new.load('test.txt')

        print(new)
        print("")
        print(conf.env)
```

The execution will result in the following output:

```
$ waf configure
'CXXFLAGS' ['-O2']
'PREFIX' '/usr/local'

'CXXFLAGS' ['-O2']
'PREFIX' '/usr/local'
'TEST' 'test'

$ cat test.txt
CXXFLAGS = ['-O2']
PREFIX = '/usr/local'

$ cat build/c4che/default.cache.py
CXXFLAGS = ['-O2']
PREFIX = '/usr/local'
TEST = 'test'
```

The default configuration is stored to the cache file named *default.cache.py*. The configuration set will always contain the default variable PREFIX, which is used to indicate the default directory in which to install something (in case there is something to install). The cache file is written in pseudo-python, and may be edited manually if necessary.

Although the conf.env object is similar to a Python dict, a few routines are present to ease value access:

```
top = '.'
out = 'build'

def configure(conf):
        conf.env['CCFLAGS'] = ['-g'] ❶
        conf.env.CCFLAGS = ['-g'] ❷
        conf.env.append_value('CXXFLAGS', ['-O2', '-g']) ❸
        conf.env.append_unique('CCFLAGS', ['-g', '-O2']) ❹
        conf.env.prepend_value('CCFLAGS', ['-O3']) ❺
        my_copy = conf.env.copy() ❻
        print(conf.env) ❼

        import Utils
        s = Utils.subst_vars('${PREFIX}/bin', conf.env) ❽
        print(s)

        conf.env.store('/tmp/env.txt') ❾

        import Environment
        env = Environment.Environment()
        env.load('/tmp/env.txt') ❿
```

❶ Initialize and set the given value by using the key-based notation

❷ Initialize and set the given value by using the attribute-based notation

❸ Ensure conf.env.CXXFLAGS is initalized, and add the given values

❹ Append the elements from the second list that are not already present in conf.env.CCFLAGS

❺ Insert the values from the second list in first position, preserving the order

❻ Create a copy

❼ Format and print the contents

❽ Substitute the values in a formatted string

❾ Serialize and store the contents into a file, one key=value by line.

❿ Instantiate a new Environment and load the contents from a file

Upon execution, the output will be the following:

```
$ waf configure
'CCFLAGS' ['-O3', '-g', '-O2']
'CXXFLAGS' ['-O2', '-g']
'PREFIX' '/usr/local'
-O3 -g -O2
```

3.2 Consulting configuration sets during the build

By splitting the configuration from the build step, it is possible to stop and look at the configuration sets (for example `build/c4che/de`
The data used during the build should use the command-line options as little as possible. To achieve this, a pattern similar to the following is usually applied:

```
top = '.'
out = 'build'

def set_options(ctx):
        ctx.add_option('--foo', action='store', default=False, help='Silly test')

def configure(ctx):
        import Options
        ctx.env.FOO = Options.options.foo
        ctx.find_program('touch', var='TOUCH', mandatory=True) # a configuration helper

def build(ctx):
        print(ctx.env.TOUCH)
        print(ctx.env.FOO)
```

All configuration sets are automatically loaded during the build. Upon execution, the output will be similar to the following:

```
$ waf distclean configure build --foo=abcd
'distclean' finished successfully (0.002s)
Checking for program touch              : ok /usr/bin/touch
'configure' finished successfully (0.001s)
Waf: Entering directory `/comp/waf/demos/simple_scenarios/folders/build'
/usr/bin/touch
abcd
Waf: Leaving directory `/comp/waf/demos/simple_scenarios/folders/build'
'build' finished successfully (0.003s)
```

3.3 Launching and catching configuration exceptions

Configuration helpers are methods provided by the conf object to help finding parameters, for example the method *conf.find_program*

```
def configure(conf):
        conf.find_program('test')
```

When a test fails, the exception *Configure.ConfigurationError* is raised. For example, such exceptions are thrown when a program could not be found:

```
def configure(conf):
        import Configure
        try:
                conf.find_program('missing_app', mandatory=True)
        except Configure.ConfigurationError:
                pass
        conf.find_program(['gjc', 'cjg'], mandatory=True)
```

The output will be:

```
$ waf
Checking for program missing_app         : not found
Checking for program gjc,cjg             : not found
 error: The program ['gjc', 'cjg'] could not be found
```

It is sometimes useful to raise such an exception manually by using *conf.fatal*:

```
def configure(conf):
        conf.fatal("I'm sorry Dave, I'm afraid I can't do that")
```

Which will display the same kind of error

```
$ waf configure
 error: I'm sorry Dave, I'm afraid I can't do that
$ echo $?
1
```

3.4 Loading Waf tools

For efficiency reasons, the Waf core libraries do not provide support for programming languages. The configuration helpers are located in separate files called *Waf tools* and located under the folder `wafadmin/Tools`. The contributed tools, or the tools in testing phase are located under the folder `wafadmin/3rdparty`.

Tools usually provide additional routines, and may modify existing classes to extend their behaviour. We will now demonstrate a very simple Waf tool named `dang.py`:

```
from Configure import conftest ❶

@conftest
def detect(conf): ❷
        print('→ detect from dang!')

@conftest
def test_hello(conf):
        print('→ test from dang!')

def build_hello(self):
        print('→ build hello from dang!')

import Build
Build.BuildContext.build_hello = build_hello ❸
```

❶ The decorator *conftest* is used to bind functions as new methods to the *configuration context class*

❷ The default detection method to execute when the tool is loaded is called *detect*

❸ New methods may be bound at runtime onto various classes, here the build context class

For loading a tool, the method *check_tool* must be used during the configuration. The same tools are then loaded during the build.:

```
top = '.'
out = 'build'

#import config_c ❶

def configure(conf):

        #import config_c ❷

        def is_check_cc_present():
                try:
                        print('→ method conf.check_cc %r' % conf.check_cc) ❸
                except:
                        print('→ there is no method conf.check_cc')
        is_check_cc_present()
        conf.check_tool('config_c') ❹
        is_check_cc_present()

        conf.check_tool('dang', tooldir='.') ❺
        conf.check_tool('dang', tooldir='.', funs='test_hello') ❻
        conf.check_tool('dang', tooldir='.') ❼

def build(bld):
        bld.build_hello() ❽
```

❶ By loading a Waf tool at the root of a script, the detection routines will not be executed (discouraged)

❷ By loading a Waf tool using *import*, the tool will not be loaded at build time (discouraged)

❸ Just an example to show if the method *check_cc* is present on the configuration context class

❹ Load the tool providing configuration routines for c and c++ support (the method check_cc)

❺ When tools provide a method named *detect*, it is executed during *conf.check_tool*

❻ Use specific functions instead of calling the default

❼ To ease the creation of projects split into modules, conf.check_tool will not load the tools twice for the same environment and the same parameters.

❽ Use the build context method defined in dang.py

Upon execution, the output will be the following:

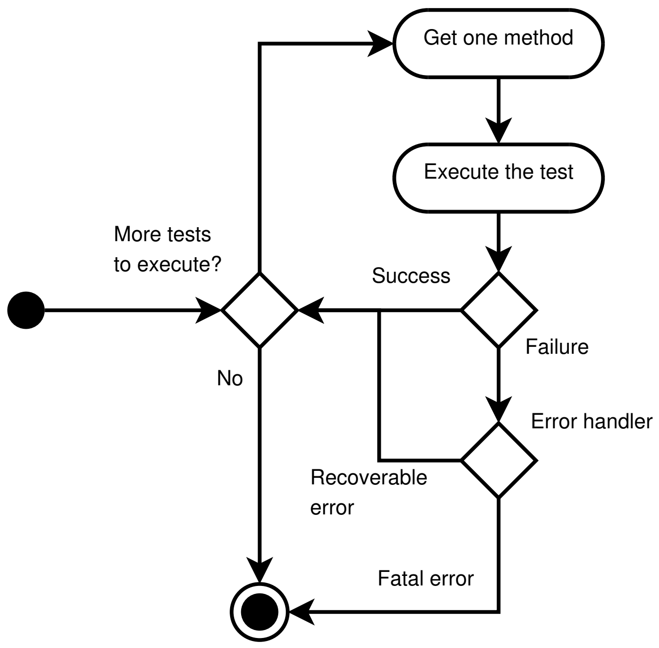
```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
→ there is no method conf.check_cc
→ method conf.check_cc (ConfigurationContext.check_cc)
→ detect from dang!
→ test from dang!
'configure' finished successfully (0.003s)
Waf: Entering directory `/tmp/smallproject/build'
→ build hello from dang!
Waf: Leaving directory `/tmp/smallproject/build'
'build' finished successfully (0.003s)
```

3.5 Handling configuration exceptions

An error handler attached to the conf object is used for catching the Configuration exceptions and processing the errors. Here is how to replace the default configuration error handler by a custom method which may modify the list of tests, stop the evaluation, or re-raise the exception:

```
import Constants
from Configure import conf

@conf
def error_handler(fun, exc):
        print('exception %r' % exc)
        # other optionals return values: Constants.CONTINUE or anything else to re-raise ↩
           the exception
        return Constants.BREAK
```

The following diagram illustrates the test execution loop performed from conf.check_tool



Chapter 4

The build phase

The build phase consists in generating the project outputs as build files. It is usually executed after the configuration phase.

4.1 Declaring new targets

Since the Waf commands build, clean, install and uninstall execute the same function *build*, it is somehow necessary to isolate the declaration of the targets from the actual code that will create them. For example:

1. Execution control: targets are only produced during the calls to the commands *build* and *install*
2. Parallel target processing to accelerate the builds: multi-core systems
3. Filesystem abstraction: fetching data from the network
4. Extensibility: adding support for new compilers and new languages from user scripts
5. Flexibility: changing the semantics of the declarations depending on the platform (operating system, compiler, etc)

These main contraints lead to the creation of an abstraction layer between the actual code execution (*task*) and the declaration (*task generators*). Here are two important definitions:

1. `Task`: Abstract unit of data transformation which may be delegated for later execution. Task instances often present sequential constraints and require a scheduler to optimize the overall execution time (execution in parallel whenever possible).
2. `Task generator`: Object part of the user interface (Waf scripts) used to create lots of task instances. The task generators handle global constraints across the tasks: access to the configuration sets, data sharing, operating system abstraction, error checking, etc

Here is a sample project script to illustrate the concepts:

```
top = '.'
out = 'build'

def configure(ctx):
        pass

def build(ctx):
        obj = ctx(target='test.txt', rule='touch ${TGT}')
        print('  ' + obj.__class__)
```

The call *ctx(...)* is just a shortcut on the build context to create new task generator instances easily. The output will be the following:

```
$ waf distclean configure build build clean
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/smallproject/build'
  class 'TaskGen.task_gen' ❶
[1/1] test.txt:  -> build/default/test.txt ❷
Waf: Leaving directory `/tmp/smallproject/build'
'build' finished successfully (0.011s)
Waf: Entering directory `/tmp/smallproject/build'
  class 'TaskGen.task_gen' ❸
Waf: Leaving directory `/tmp/smallproject/build'
'build' finished successfully (0.004s)
  class 'TaskGen.task_gen' ❹
'clean' finished successfully (0.002s)
```

❶ Print *obj*.class.

❷ Actual task execution, during the first build.

❸ This print was output during the second `build` execution. The target is considered up-to-date and is not rebuilt.

❹ This print was output during the call to `clean`.

4.2 Finding folders and files through node objects

In general, file system access are slow operations (listing files and folders, reading file permissions, etc). For this reason, Waf maintains an internal representation of the files and folders already explored in the form of a tree of Node instances. Two nodes may be accessed from the build context:

1. `root`: node representing the root of the file system, or the folder containing the drive letters on win32 systems
2. `path`: node representing the path of the script being read (path to the wscript file)

Nodes are often used for low-level operations such as extending Waf, and direct node manipulation is hardly ever necessary from user scripts.

```
top = '.'
out = 'build'

def configure(ctx):
        pass

def build(bld):
        print('current path %r' % bld.path)
        print('absolute path of the root node: %r' % bld.root.abspath())
        print('absolute path of the current node: %r' % bld.path.abspath())

        etc = bld.root.find_dir('/etc')
        var = bld.root.find_dir('/var')
        print('the node representing /etc %r: ' % etc)
        print('path from /var to /etc %r' % etc.relpath_gen(var))

        fstab = bld.root.find_resource('/etc/fstab')
        print('path to /etc/fstab: %r' % fstab.abspath())
```

The following results would be observed upon execution on a unix-like system:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/smallproject/build'
current path dir:///tmp/smallproject
absolute path of the root node: '/'
absolute path of the current node: '/tmp/smallproject'
the node representing /etc dir:///etc:
path from /var to /etc '../etc'
path to /etc/fstab: '/etc/fstab'
Waf: Leaving directory `/tmp/smallproject/build'
'build' finished successfully (0.003s)
```

Listing files and folders automatically reduces the needs for updating the scripts (files may be added, removed or renamed). A Node method called *ant_glob* enables searching for folders and files on the file system. Here is an illustration:

```
top = '.'
out = 'build'

def configure(ctx):
        pass

def build(bld):
        print(bld.path.ant_glob('wsc*'))❶
        print(bld.path.ant_glob('w?cr?p?'))❷
        print(bld.root❸.ant_glob('etc/**/g*'❹, dir=True, src=False, bld=False)) ❺
```

❶ The method ant_glob is called on a node object, and not on the build context

❷ The patterns may contain wildcards such as * or ?, but they are Ant patterns, not regular expressions

❸ Calling ant_glob on the file system root may be slow, and may give different results depending on the operating system

❹ The ** indicates to consider folders recursively. Use with care.

❺ By default, only node representing source files are returned. It is possible to obtain folder nodes and build file nodes by turning on the appropriate options.

The execution output will be similar to the following:

```
$ waf
Waf: Entering directory `/tmp/smallproject/build'
wscript
wscript
etc/ghostscript etc/gconf etc/gconf/gconf.xml.mandatory etc/gnome-vfs-2.0 etc/gpm etc/gnupg ↩
     etc/gre.d etc/gdm
Waf: Leaving directory `/tmp/smallproject/build'
'build' finished successfully (12.873s)
```

4.3 Installing files

Three build context methods are provided for installing files:

1. install_files: install several files in a folder
2. install_as: install a target with a different name
3. symlink_as: create a symbolic link on the platforms that support it

```
def build(bld):
        bld.install_files('${PREFIX}/include', ['a1.h', 'a2.h']) ❶
        bld.install_as('${PREFIX}/dir/bar.png', 'foo.png') ❷
        bld.symlink_as('${PREFIX}/lib/libfoo.so.1', 'libfoo.so.1.2.3') ❸

        env_foo = bld.env.copy()
        env_foo.PREFIX = '/opt'
        bld.install_as('${PREFIX}/dir/test.png', 'foo.png', env=env_foo) ❹

        start_dir = bld.path.find_dir('src/bar')
        bld.install_files('${PREFIX}/share', ['foo/a1.h'], cwd=start_dir, relative_trick= ↩
            True) ❺

        bld.install_files('${PREFIX}/share', start_dir.ant_glob('**/*.png'), cwd=start_dir, ↩
             relative_trick=True) ❻
```

❶ Install various files in the target destination

❷ Install one file, changing its name

❸ Create a symbolic link

❹ Overridding the configuration set (*env* is optional in the three methods install_files, install_as and symlink_as)

❺ Install src/bar/foo/a1.h as seen from the current script into *${PREFIX}/share/foo/a1.h*

❻ Install the png files recursively, preserving the folder structure read from src/bar/

4.4 Executing specific routines before or after the build

User functions may be bound to be executed at two key moments during the build command (callbacks):

1. immediately before the build starts (bld.add_pre_fun)
2. immediately after the build is completed successfully (bld.add_post_fun)

Here is how to execute a test after the build is finished:

```
top = '.'
out = 'build'

def set_options(ctx):
        ctx.add_option('--exe', action='store_true', default=False,
                help='execute the program after it is built')

def configure(ctx):
        pass

def pre(ctx): ❶
        print('before the build is started')

def post(ctx):
        import Options, Utils
        if Options.options.exe: ❷
                Utils.exec_command('/sbin/ldconfig') ❸

def build(ctx):
        ctx.add_pre_fun(pre) ❹
        ctx.add_post_fun(post)
```

1. The callbacks take the build context as unique parameter *ctx*
2. Access to the command-line options
3. A common scenario is to call ldconfig after the files are installed.
4. Scheduling the functions for later execution. Remember that in Python, functions are objects too.

Upon execution, the following output will be produced:

```
$ waf distclean configure build --exe
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/smallproject/build'
before the build is started (1)
Waf: Leaving directory `/tmp/smallproject/build'
/sbin/ldconfig: Can't create temporary cache file /etc/ld.so.cache~: Permission denied (2)
'build' finished successfully (0.008s)
```

1. output of the function bound by *bld.add_pre_fun*
2. output of the function bound by *bld.add_post_fun*

Chapter 5

Build copies and variants

A common scenario is to duplicate the outputs for a particular build phase. The copy may be performed into another build directory, or into subfolders of the same tree called variants.

5.1 Using several build folders

It is not possible to use several Waf instances concurrently over the same build folder. Yet, several Waf instances may use the project at the same time. For this, two options must be set:

1. The environment variable `WAFCACHE`
2. The build directory, using a command-line option

Here is an example for a simple project located in */tmp/smallfolderr*‘:

```
top = '.'
out = 'out_directory'

def configure(conf):
        pass

def build(bld):
        bld(rule='touch ${TGT}', target='foo.txt')
```

Upon execution, the results will be the following:

```
$ export WAFLOCK=.lock-debug ❶

$ waf distclean configure -b debug ❷
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.001s)

$ waf
Waf: Entering directory `/tmp/smallproject/debug'
[1/1] foo.txt:  -> debug/default/foo.txt ❸
Waf: Leaving directory `/tmp/smallproject/debug'
'build' finished successfully (0.012s)

$ export WAFLOCK=.lock-release

$ waf distclean configure -b release
'distclean' finished successfully (0.001s)
```

```
'configure' finished successfully (0.176s)

$ waf
Waf: Entering directory `/tmp/smallproject/release' ④
[1/1] foo.txt:  -> release/default/foo.txt
Waf: Leaving directory `/tmp/smallproject/release'
'build' finished successfully (0.034s)

$ tree -a
.
|-- .lock-debug ⑤
|-- .lock-release
|-- debug
|   |-- .wafpickle-7
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- default.cache.py
|   |-- config.log
|   `-- default
|       `-- foo.txt
|-- release
|   |-- .wafpickle-7
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- default.cache.py
|   |-- config.log
|   `-- default
|       `-- foo.txt
`-- wscript
```

1. The environment variable *WAFLOCK* points at the configuration of the project in use.
2. The lockfile is created during the configuration.
3. The files are output in the build directory `debug`
4. The configuration *release* is used with a different lock file and a different build directory.
5. The contents of the project directory contain the two lock files and the two build folders.

When waf is executed, it reads the variable *WAFLOCK* on an internal variable, which may be modified programmatically:

```
import Options
Options.lockfile = '.lockfilename'
```

5.2 Defining variants

Using different build folders is very useful for checking at some point if a different configuration would compile properly. To create different kinds of builds at once, it is possible to use *Waf variants* to predefine the configuration sets for specific output subdirectories.

We will now demonstrate the definition and the usage of two variants named *default* and *debug* respectively:

```
top = '.'
out = 'out_bld'

def configure(conf):
        conf.env.NAME = 'default'
        dbg = conf.env.copy() ①
        rel = conf.env.copy()

        dbg.set_variant('debug') ❷
        conf.set_env_name('debug', dbg) ❸
        conf.setenv('debug') ❹
        # conf.check_tool('gcc') ❺
        conf.env.NAME = 'foo' ❻

        rel.set_variant('release') ❼
        conf.set_env_name('cfg_name', rel)
        conf.setenv('cfg_name')
        conf.env.NAME = 'bar'

def build(bld):
        bld(
                rule='echo ${NAME} > ${TGT}',
                target='test.txt')

        bld(
                rule='echo ${NAME} > ${TGT}',
                target='test.txt'❽,
                env=bld.env_of_name('debug').copy()) ❾

        bld(
                rule='echo ${NAME} > ${TGT}',
                target='test.txt',
                env=bld.env_of_name('cfg_name').copy())
```

❶ Create a copy of the default data set.

❷ Set the copy to use the variant named *debug*: task using it will output their files into `out_bld/debug`

❸ Bind the configuration set to the configuration. The configuration set will be saved when the configuration terminates

❹ Replace *conf.env* by our new debug configuration set

❺ Waf tools store their configuration data on conf.env, in this case the *debug* configuration set, not in the default

❻ Store a variable on the *debug* configuration set

❼ Define another variant called *release*. The variant name and the configuration set name may be different.

❽ The argument *env* is given to specify the task generator configuration set. The configuration set holds the variant definition.

❾ Environments may be retrieved by name from the build context object. It is recommended to make copies to avoid accidental data sharing.

Upon execution, an output similar to the following will be observed:

```
$ waf distclean configure build
'distclean' finished successfully (0.000s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/smallproject/out_bld'
[1/3] test.txt:  -> out_bld/default/test.txt
[2/3] test.txt:  -> out_bld/debug/test.txt ❶
[3/3] test.txt:  -> out_bld/release/test.txt
Waf: Leaving directory `/tmp/smallproject/out_bld'
'build' finished successfully (0.020s)

$ tree out_bld/
out_bld/
|-- c4che
```

```
|    |-- cfg_name.cache.py ❷
|    |-- build.config.py
|    |-- debug.cache.py
|    `-- default.cache.py
|-- config.log
|-- debug
|    `-- test.txt
|-- default
|    `-- test.txt ❸
`-- release
     `-- test.txt

$ cat out_bld/default/test.txt out_bld/debug/test.txt out_bld/release/test.txt
default ❹
foo
bar
```

❶ The files are output in their respective variants

❷ The configuration sets are stored and retrieved by names, which must be valid filenames without blanks.

❸ The tasks output their files in the relevant variant

❹ The file contents are different and correspond to the configuration sets used

Note
As a general rule, tasks created for a particular variant should not share data with the tasks from another variant.

5.3 Cloning task generators

A cloning scheme is provided for duplicating task generators for several variants easily. A general design pattern is to duplicate the task generators for the desired variants immediately before the build starts. Here is an example, also available in the folder `demos/cxx` of the Waf distribution.

```
top = '.'
out = 'build'

import Options

def set_options(opt):
        opt.add_option('--build_kind', action='store', default='debug,release', help='build ↩
            the selected variants') ❶

def configure(conf): ❷
        for x in ['debug', 'release']:
                env = conf.env.copy()
                env.set_variant(x)
                conf.set_env_name(x, env)

def build(bld):

        bld(rule='touch ${TGT}', target='foo.txt') ❸

        for obj in bld.all_task_gen[:]: ❹
                for x in ['debug', 'release']:
                        cloned_obj = obj.clone(x) ❺
                        kind = Options.options.build_kind
```

```
                    if kind.find(x) < 0:❻
                            cloned_obj.posted = True
            obj.posted = True ❼
```

❶ Add a command-line option for enabling or disabling the *release* and *debug* builds

❷ The configuration will create the *release* and *debug* configuration sets, bound to a variant of the same names

❸ Targets are declared normally for the default variant

❹ A copy of the existing task generators is created to avoid the creation of an infinite loop (new task generator instances get added to that list)

❺ Clone a task generator for the configuration set *debug* or *release*. Making task generator clones is a cheap operation compared to duplicating tasks.

❻ Look at the command-line arguments, and disable the unwanted variant(s)

❼ Disable the original task generator for the default configuration set (do not use it).

Some task generators are use indexed attributes to generate unique values which may cause unnecessary rebuilds if the scripts change. To avoid problems, it is a best practice to create the task generators for the default configuration set first. Also, the method *clone* is not a substitute for creating instances for lots of nearly identical task generators. In such a situation, it will be better to use one task generator to create lots of tasks. As a reminder, creating task generator clones for the same variant will lead to build errors.

Warning
Do not create task generator clones for the same variant or for the same configuration set.

Chapter 6

Advanced command definitions

The waf operations (commands) may be extended to perform non-trivial operations easily. This chapter demonstrates several common patterns found in real-world project scripts.

6.1 Providing a custom command context

The context for a command is created automatically, and is derived from the class *Utils.Context*. Custom context instance may be provided in the following manner:

```
def foo(ctx):
        pass

import Utils
class foo_context(Utils.Context):
        def __init__(self):
                print("a context for 'foo'")
```

Custom contexts may be provided for the functions *configure* and *build*.

6.2 Creating aliases / Injecting new commands

New commands may be injected in the following manner:

```
import Scripting
def foo(ctx):
        Scripting.commands += ['build', 'clean']
```

Injecting new commands is useful for writing testcases. By executing *waf test*, the following script will configure a project, create source files in the source directory, build a program, modify the sources, and rebuild the program. In this case, the program must be rebuilt because a header (implicit dependency) has changed.

```
VERSION = '0.0.1'
APPNAME = 'my_testcase'

top = '.'
out = 'build'

import Scripting

def test(ctx):
        lst = 'distclean configure setup build modify build'.split()
        Scripting.commands += lst

def configure(conf):
        conf.check_tool('gcc')

def setup(ctx):
        f = open('main.c', 'w')
        f.write('#include "foo.h"\nint main() {return 0;}\n')
        f.close()

        f = open('foo.h', 'w')
        f.write('int k = 32;\n')
        f.close()

def build(bld):
        bld(
                features = 'cc cprogram',
                source = 'main.c',
                target='tprog')

def modify(ctx):
        f = open('foo.h', 'w')
        f.write('int k = 34;\n')
        f.close()
```

6.3 Executing commands as part of another command

In the following scenario, the build files are installed into a temporary directory for packaging. This is different from the `waf dist` command which only packages the source files needed for the project.

Because the context objects usually derive from different classes, it is usually forbidden to call a command from another commands. In the following example, the commands to execute are pushed onto a stack:

```
top = '.'
out = 'out'

import Scripting, Options

def configure(conf):
        conf.check_tool('gcc')

def build(bld):
        bld.install_files('/tmp/foo/', 'wscript') ❶
        bld.new_task_gen(features='cc cprogram', target='foo', source='k.c')

back = False
def package(ctx): ❷
        global back
        Scripting.commands.insert(0, 'create_package') ❸
        Scripting.commands.insert(0, 'install') ❹
        back = Options.options.destdir ❺
        Options.options.destdir = '/tmp/foo'

def create_package(ctx): ❻
        global back
        print("packaging")
        Options.options.destdir = back ❼
```

1. Perform the installation
2. Users will call *waf package* to create the package
3. Postpone the actual package creation
4. Now the next command to execute is the installation, and after the installation, the package creation will be performed
5. Change the value of *destdir* for installing the files into a temporary directory
6. Command for creating the package
7. Restore the value of *destdir*, in case if more commands are to be executed

Note that the *destdir* parameter is being passed between commands by means of a global option.

6.4 Adding a new command from a waf tool

When the top-level wscript is read, it is converted into a python module and kept in memory. To add a new command dynamically, it is only necessary to inject the desired function into that module. We will now show how to load a waf tool to count the amount of task generators in the project.

The waf tools are loaded once during the configuration and during the build. To ensure that the tool is always loaded, it is necessary to load its options:

```
top = '.'
out = 'build'

def set_options(opt):
        opt.tool_options('some_tool', tooldir='.')

def configure(conf):
        pass

def build(bld):
        bld(rule='echo "hi"', always=True, name='a')
        bld(rule='echo "there"', always=True, name='b')
```

Now our tool *some_tool.py*, located next to the *wscript* file, will contain the following code:

```
import os
import Utils, Environment, Options, Build

def set_options(opt): ❶
        pass

def list_targets(ctx): ❷
        """return the amount of targets"""

        bld = Build.BuildContext() ❸
        proj = Environment.Environment(Options.lockfile) ❹
        bld.load_dirs(proj['top'], proj['out']) ❺
        bld.load_envs()

        bld.add_subdirs([os.path.split(Utils.g_module.root_path)[0]]) ❻

        names = set([])
        for x in bld.all_task_gen: ❼
                try:
                        names.add(x.name or x.target)
                except AttributeError:
                        pass

        lst = list(names)
        lst.sort()
        for name in lst:
                print(name)
Utils.g_module.__dict__['list_targets'] = list_targets ❽
```

1. The function *set_options* is called by *tool_options* above
2. The command that will be injected dynamically from this tool
3. Declare a build context manually
4. Read the project settings
5. Initialize the build context
6. Read the project files, executing the methods *build*
7. Access the declared task generators stored in the build context attribute *all_task_gen*
8. Bind the function as a new command

The execution output will be the following.

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/home/waf/tmp/nt/build'
[1/2] echo "hi":
hi
[2/2] echo "there":
there
Waf: Leaving directory `/home/waf/tmp/nt/build'
'build' finished successfully (0.015s)

$ waf list_targets
a
b
'list_targets' finished successfully (0.003s)

$ waf --targets=a
Waf: Entering directory `/home/waf/tmp/nt/build'
[1/1] a:
hi
Waf: Leaving directory `/home/waf/tmp/nt/build'
'build' finished successfully (0.012s)
```

Note that during the execution of `num_targets`, the task generator definitions have been read but no task was executed.

Chapter 7

Rule-based task generators (Make-like)

This chapter illustrates how to perform common transformations used during the build phase through the use of rule-based task generators.

7.1 Declaration and usage

Rule-based task generators are a particular category of task generators producing exactly one task (transformation) at a time.

The following example presents a simple example of a task generator producing the file *foobar.txt* from the project file *wscript* by executing a copy (the command `cp`). Let's create a new project in the folder */tmp/rule/* containing the following *wscript* file:

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        bld( ❶
                rule   = 'cp ${SRC} ${TGT}', ❷
                source = 'wscript', ❸
                target = 'foobar.txt', ❹
        )
```

❶ To instantiate a new task generator, remember that all arguments have the form *key=value*

❷ The attribute *rule* is mandatory here. It represents the command to execute in a readable manner (more on this in the next chapters).

❸ Source files, either in a space-delimited string, or in a list of python strings

❹ Target files, either in a space-delimited string, or in a list of python strings

Upon execution, the following output will be observed:

```
$ waf distclean configure build -v
'distclean' finished successfully (0.000s)
'configure' finished successfully (0.021s)
Waf: Entering directory `/tmp/rule/out'
[1/1] foobar.txt: wscript -> out/default/foobar.txt ❶
16:24:21 runner system command ->  cp ../wscript default/foobar.txt ❷
Waf: Leaving directory `/tmp/rule/out'
```

```
'build' finished successfully (0.016s)

$ tree
.
|-- out
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- default.cache.py
|   |-- config.log
|   `-- default
|       `-- foobar.txt
`-- wscript

$ waf ❸
Waf: Entering directory `/tmp/rule/out'
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.006s)

$ echo " " >> wscript ❹

$ waf
Waf: Entering directory `/tmp/rule/out'
[1/1] foobar.txt: wscript → out/default/foobar.txt ❺
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.013s)
```

❶ In the first execution, the target is correctly created

❷ Command-lines are only displayed in *verbose mode* by using the option *-v*

❸ The target are up-to-date, there is nothing to do

❹ Modify the source file in place by appending a space character

❺ Since the source has changed, the target is created once again.

The target is created once again whenever the source files or the rule change. This is achieved by computing a signature for the targets, and storing that signature between executions. By default, the signature is computed by hashing the rule and the source files (MD5 by default).

Note
The task (or transformation) are only executed during the build phase, after all build functions have been read

7.2 Rule functions

Rules may be given as expression strings or as python function. Let's modify the previous project file with a python function:

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        def run(task): ❶
                print(' → source is ' + task.generator.source) ❷
                src = task.inputs[0].srcpath(task.env) ❸
                tgt = task.outputs[0].bldpath(task.env) ❹

                import Utils
                cmd = 'cp %s %s' % (src, tgt)
                print(cmd)
                return Utils.exec_command(cmd) ❺

        bld(
                rule   = run, ❻
                source = 'wscript',
                target = 'same.txt',
        )
```

❶ Rule functions take the task instance as parameter.

❷ Task instances may access their task generator through the attribute *generator*

❸ Sources and targets are represented internally as Node objects bound to the task instance.

❹ Commands are executed from the root of the build directory. Node methods such as *bldpath* ease the command line creation.

❺ Utils.exec_command(...) is a wrapper around subprocess.Popen(...) from the Python library. Passing a string will execute the command through the system shell (use lists to disable this behaviour). The return code for a rule function must be non-0 to indicate a failure.

❻ Use a function instead of a string expression

The execution trace will be similar to the following:

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/rule/out'
[1/1] same.txt: wscript -> out/default/same.txt
 → source is wscript
cp ../wscript default/same.txt
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.010s)
```

The rule function must return *0* to indicate success, and must generate the files corresponding to the outputs. The rule function must also access the task object in a read-only manner, and avoid node creation or attribute modification.

Note
The string expression *cp ${SRC} ${TGT}* from the previous example is converted internally to a function similar to *run*.

Note
While a string expression may execute only one system command, functions may execute various commands at once.

Warning
Due to limitations in the cPython interpreter, only functions defined in python modules can be hashed. This means that changing a function will trigger a rebuild only if it is defined in a waf tool (not in wscript files).

7.3 Shell usage

The attribute *shell* is used to enable the system shell for command execution. A few points are worth keeping in mind when declaring rule-based task generators:

1. The Waf tools do not use the shell for executing commands
2. The shell is used by default for user commands and custom task generators
3. String expressions containing the following symbols ’>’, ’<’ or ’&’ cannot be transformed into functions to execute commands without a shell, even if told to
4. In general, it is better to avoid the shell whenever possible to avoid quoting problems (paths having blank characters in the name for example)
5. The shell is creating a performance penalty which is more visible on win32 systems.

Here is an example:

```
top = ’.’
out = ’out’

def configure(conf):
        pass

def build(bld):
        bld(rule=’cp ${SRC} ${TGT}’, source=’wscript’, target=’f1.txt’, shell=False)
        bld(rule=’cp ${SRC} ${TGT}’, source=’wscript’, target=’f2.txt’, shell=True)
```

Upon execution, the results will be similar to the following:

```
waf distclean configure build --zones=runner,action ❶
’distclean’ finished successfully (0.004s)
’configure’ finished successfully (0.001s)
Waf: Entering directory ‘/tmp/rule/out’
23:11:23 action ❷
def f(task):
        env = task.env
        wd = getattr(task, ’cwd’, None)
        def to_list(xx):
                if isinstance(xx, str): return [xx]
                return xx
        lst = []
        lst.extend([’cp’])
        lst.extend([a.srcpath(env) for a in task.inputs])
        lst.extend([a.bldpath(env) for a in task.outputs])
        lst = [x for x in lst if x]
        return task.exec_command(lst, cwd=wd)

23:11:23 action
def f(task): ❸
        env = task.env
        wd = getattr(task, ’cwd’, None)
        p = env.get_flat
        cmd = ’’’ cp %s %s ’’’ % (" ".join([a.srcpath(env) for a in task.inputs]),
                " ".join([a.bldpath(env) for a in task.outputs]))
        return task.exec_command(cmd, cwd=wd)

[1/2] f1.txt: wscript -> out/default/f1.txt
23:11:23 runner system command -> [’cp’, ’../wscript’, ’default/f1.txt’] ❹
[2/2] f2.txt: wscript -> out/default/f2.txt
```

```
23:11:23 runner system command ->  cp ../wscript default/f2.txt
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.017s)
```

1. The *debugging zones* enable the display of specific debugging information (comma-separated values) for the string expression conversion, and *runner* for command execution
2. String expressions are converted to functions (here, without the shell).
3. Command execution by the shell. Notice the heavy use of string concatenation.
4. Commands to execute are displayed by calling *waf --zones=runner*. When called without the shell, the arguments are displayed as a list.

Note
Whenever possible, avoid using the shell to improve both performance and maintainability

7.4 Inputs and outputs

Source and target arguments are optional for make-like task generators, and may point at one or several files at once. Here are a few examples:

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        bld( ❶
                rule   = 'cp ${SRC} ${TGT[0].abspath(env)} && cp ${SRC} ${TGT[1].abspath( ↩
                    env)}',
                source = 'wscript',
                target = 'f1.txt f2.txt',
                shell  = True
        )

        bld( ❷
                source = 'wscript',
                rule   = 'echo ${SRC}'
        )

        bld( ❸
                target = 'test.k3',
                rule   = 'echo "test" > ${TGT}',
        )

        bld( ❹
                rule   = 'echo 1337'
        )

        bld( ❺
                rule   = "echo 'task always run'",
                always = True
        )
```

1. Generate *two files* whenever the input or the rule change. Likewise, a rule-based task generator may have multiple input files.
2. The command is executed whenever the input or the rule change. There are no declared outputs.
3. No input, the command is executed whenever it changes
4. No input and no output, the command is executed only when the string expression changes
5. No input and no output, the command is executed each time the build is called

For the record, here is the output of the build:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.093s)
Waf: Entering directory `/tmp/rule/out'
[1/5] echo 1337:
1337
[2/5] echo 'task always run':
[3/5] echo ${SRC}: wscript
../wscript
[4/5] f1.txt f2.txt: wscript -> out/default/f1.txt out/default/f2.txt
task always run
[5/5] test.k3:  -> out/default/test.k3
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.049s)

$ waf
Waf: Entering directory `/tmp/rule/out'
[2/5] echo 'task always run':
task always run
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.014s)
```

7.5 Sharing data

Data sharing is performed through the configuration set. The following example illustrates how to use it.

```
top = '.'
out = 'out'

import Options

def configure(conf):
        print('prefix: %r' % conf.env.PREFIX)
        print('jobs:   %r' % Options.options.jobs)

def build(bld):
        bld(
                target = 't1.txt',
                rule   = 'echo ${PREFIX} > ${TGT}' ❶
        )

        obj = bld(
                target = 't2.txt',
                rule   = 'echo ${TEST_VAR} > ${TGT}', ❷
        )
        obj.env.TEST_VAR ❸= str(Options.options.jobs) ❹
        bld(
                rule = 'echo "hey"',
                vars = ['TEST_VAR'], ❺
                env  = obj.env ❻
        )
```

❶ The *PREFIX* is one of the few predefined variables. It is necessary for computing the default installation path.

❷ The following rule will create the file *t2.txt* with the contents of *TEST_VAR*

❸ The value of *TEST_VAR* will be defined now

❹ Use the value of a predefined command-line option (the jobs control the amount of commands which may be executed in parallel)

❺ By default, the variables used in string expressions *${. . . }* are extracted automatically and used as dependencies (rebuild the targets when the value change). They are given manually in this case.

❻ Set the base environment. The variable TEST_VAR will be used for the dependency here.

By turning the debugging flags on, it will be easier to understand what is happening during the build:

```
$ waf distclean configure build --zones=runner,action
'distclean' finished successfully (0.003s)
prefix: '/usr/local' ❶
jobs:   2
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/rule/out'
15:21:29 action
def f(task):
        env = task.env
        wd = getattr(task, 'cwd', None)
        p = env.get_flat
        cmd = ''' echo %s > %s ''' % (p('PREFIX'), ❷
                " ".join([a.bldpath(env) for a in task.outputs]))
        return task.exec_command(cmd, cwd=wd)

[...] ❸

[1/3] t2.txt:  -> out/default/t2.txt
15:21:29 runner system command ->  echo 2 > default/t2.txt
[2/3] t1.txt:  -> out/default/t1.txt
15:21:29 runner system command ->  echo /usr/local > default/t1.txt ❹
[2/3] echo "hey":
18:05:26 runner system command ->  echo "hey"
hey
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.052s)

$ waf
Waf: Entering directory `/tmp/rule/out'
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.007s)

$ waf --jobs=17 --zones=runner
Waf: Entering directory `/tmp/rule/out'
[1/3] t2.txt:  -> out/default/t2.txt
15:23:24 runner system command ->  echo 17 > default/t2.txt ❺
[2/3] echo "hey": ❻
hey
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.014s)
```

1. The default values for prefix and jobs.
2. The function generated from the string expression will use the *PREFIX* value.
3. Some output was removed.
4. The expression *${PREFIX}* was substituted by its value.
5. The target is created whenever the value of a variable changes, even though the string expression has not changed.
6. The variable *TEST_VAR* changed, so the command was executed again.

7.6 Execution order and dependencies

Although task generators will create the tasks in the relevant order, tasks are executed in parallel and the compilation order may not be the order expected. In the example from the previous section, the target *t2.txt* was processed before the target *t1.txt*. We will now illustrate two important concepts used for the build specification:

1. order: sequential constraints between the tasks being executed (here, the commands)
2. dependency: executing a task if another task is executed

For illustation purposes, let's create the following chain *wscript* → *w1.txt* → *w2.txt*. The order constraint is given by the attribute *after* which references a task class name. For rule-based task generators, the task class name is bound to the attribute *name*

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        bld(
                rule   = 'cp ${SRC} ${TGT}',
                source = 'wscript',
                target = 'w1.txt',
                name   = 'w1',
        )

        bld(
                rule   = 'cp ${SRC} ${TGT}',
                source = 'w1.txt',
                target = 'w2.txt'
                after  = w1,
        )
```

The execution output will be similar to the following. Note that both files are created whenever the first source file *wscript* changes:

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/rule/out'
[1/2] w1: wscript -> out/default/w1.txt
[2/2] w2.txt: out/default/w1.txt -> out/default/w2.txt
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.055s)

$ echo " " >> wscript
$ waf
Waf: Entering directory `/tmp/rule/out'
[1/2] w1: wscript -> out/default/w1.txt
[2/2] w2.txt: out/default/w1.txt -> out/default/w2.txt
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.018s)
```

Although the order constraint between w1 and w2 is trivial to find in this case, implicit constraints can be a source of confusion for large projects. For this reason, the default is to *encourage* the explicit order declaration. Nevertheless, a special method may be used to let the system look at all source and target files:

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):

        bld.use_the_magic()

        bld(
                rule   = 'cp ${SRC} ${TGT}',
                source = 'wscript',
                target = 'w1.txt',
        )

        bld(
                rule   = 'cp ${SRC} ${TGT}',
                source = 'w1.txt',
                target = 'w2.txt',
        )
```

Warning
The method *bld.use_the_magic()* will not work for tasks that do not have clear input and output files and will degrade performance for large builds.

7.7 Dependencies on file contents

As a second example, we will create a file named *r1.txt* from the current date. It will be updated each time the build is executed. A second file named *r2.txt* will be created from *r1.txt*.

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        bld(
                name   = 'r1', ❶
                target = 'r1.txt',
                rule   = '(date > ${TGT}) && cat ${TGT}', ❷
                always = True, ❸
        )
        bld(
                name   = 'r2', ❹
                target = 'r2.txt',
                rule   = 'cp ${SRC} ${TGT}',
                source = 'r1.txt', ❺
                after  = 'r1', ❻
        )
```

❶ Give the task generator a name, it will create a task class of the same name to execute the command

❷ Create *r1.txt* with the date

❸ There is no source file to depend on and the rule never changes. The task is then set to be executed each time the build is started by using the attribute *always*

❹ If no name is provided, the rule is used as a name for the task class

❺ Use *r1.txt* as a source for *r2.txt*. Since *r1.txt* was declared before, the dependency will be added automatically (*r2.txt* will be re-created whenever *r1.txt* changes)

❻ Set the command generating *r2.txt* to be executed after the command generating *r1.txt*. The attribute *after* references task class names, not task generators. Here it will work because rule-based task generator tasks inherit the *name* attribute

The execution output will be the following:

```
$ waf distclean configure build -v
'distclean' finished successfully (0.003s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/rule/out'
[1/2] r1:  -> out/default/r1.txt
16:44:39 runner system command ->  (date > default/r1.txt) && cat default/r1.txt
dom ene 31 16:44:39 CET 2010
[2/2] r2: out/default/r1.txt -> out/default/r2.txt
16:44:39 runner system command ->  cp default/r1.txt default/r2.txt
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.021s)

$ waf -v
Waf: Entering directory `/tmp/rule/out'
[1/2] r1:  -> out/default/r1.txt
16:44:41 runner system command ->  (date > default/r1.txt) && cat default/r1.txt
dom ene 31 16:44:41 CET 2010
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.016s)
```

Although r2 **depends** on *r1.txt*, r2 was not executed in the second build. As a matter of fact, the signature of the task r1 has not changed, and r1 was only set to be executed each time, regardless of its signature. Since the signature of the *r1.txt* does not change, the signature of r2 will not change either, and *r2.txt* is considered up-to-date.

We will now illustrate how to make certain that the outputs reflect the file contents and trigger the rebuild for dependent tasks by enabling the attribute *on_results*:

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        bld(
                name   = 'r1',
                target = 'r1.txt',
                rule   = '(date > ${TGT}) && cat ${TGT}',
                always = True,
                on_results = True,
        )

        bld(
                target = 'r2.txt',
                rule   = 'cp ${SRC} ${TGT}',
                source = 'r1.txt',
                after  = 'r1',
        )
```

Here *r2.txt* will be re-created each time:

```
$ waf distclean configure build -v
'distclean' finished successfully (0.003s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/rule/out'
[1/2] r1:  -> out/default/r1.txt
16:59:49 runner system command ->  (date > default/r1.txt) && cat default/r1.txt ❶
dom ene 31 16:59:49 CET 2010 ❷
[2/2] r2: out/default/r1.txt -> out/default/r2.txt
16:59:49 runner system command ->  cp default/r1.txt default/r2.txt
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.020s)

$ waf -v
Waf: Entering directory `/tmp/rule/out'
[1/2] r1:  -> out/default/r1.txt
16:59:49 runner system command ->  (date > default/r1.txt) && cat default/r1.txt
dom ene 31 16:59:49 CET 2010 ❸
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.016s)

$ waf -v
Waf: Entering directory `/tmp/rule/out'
[1/2] r1:  -> out/default/r1.txt
16:59:53 runner system command ->  (date > default/r1.txt) && cat default/r1.txt
dom ene 31 16:59:53 CET 2010 ❹
[2/2] r2: out/default/r1.txt -> out/default/r2.txt
16:59:53 runner system command ->  cp default/r1.txt default/r2.txt
Waf: Leaving directory `/tmp/rule/out'
'build' finished successfully (0.022s)
```

❶ Start with a clean build, both *r1.txt* and *r2.txt* are created

❷ Notice the date and time

❸ The second build was executed at the same date and time, so *r1.txt* has not changed, therefore *r2.txt* is up to date

❹ The third build is executed at another date and time. Since *r1.txt* has changed, *r2.txt* is created once again

Chapter 8

Name and extension-based file processing

Transformations may be performed automatically based on the file name or on the extension.

8.1 Refactoring repeated rule-based task generators into implicit rules

The explicit rules described in the previous chapter become limited for processing several files of the same kind. The following code may lead to unmaintainable scripts and to slow builds (for loop):

```
def build(bld):
        for x in 'a.lua b.lua c.lua':
                y = x.replace('.lua', '.luac')
                bld(source=x, target=y, rule='${LUAC} -s -o ${TGT} ${SRC}')
                bld.install_files('${LUADIR}', x)
```

Rather, the rule should be removed from the user script, like this:

```
def build(bld):
        bld(source='a.lua b.lua c.lua')
```

The equivalent logic may then be provided by using the following code. It may be located in either the same *wscript*, or in a waf tool:

```
import TaskGen
TaskGen.declare_chain(
        name      = 'luac', ❶
        rule      = '${LUAC} -s -o ${TGT} ${SRC}', ❷
        shell     = False,
        ext_in    = '.lua', ❸
        ext_out   = '.luac', ❹
        reentrant = False, ❺
        install   = '${LUADIR}', ❻
)
```

❶ The name for the corresponding task class to use

❷ The rule is the same as for any rule-based task generator

❸ Input file, processed by extension

❹ Output files extensions separated by spaces. In this case there is only one output file

❺ The reentrant attribute is used to add the output files as source again, for processing by another implicit rule

❻ String representing the installation path for the output files, similar to the destination path from *bld.install_files*. To disable installation, set it to False.

8.2 Chaining more than one command

Now consider the long chain *uh.in* → *uh.a* → *uh.b* → *uh.c*. The following implicit rules demonstrate how to generate the files while maintaining a minimal user script:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(source='uh.in')

import TaskGen
TaskGen.declare_chain(name='a', rule='cp ${SRC} ${TGT}', ext_in='.in', ext_out='.a',)
TaskGen.declare_chain(name='b', rule='cp ${SRC} ${TGT}', ext_in='.a',  ext_out='.b',)
TaskGen.declare_chain(name='c', rule='cp ${SRC} ${TGT}', ext_in='.b',  ext_out='.c',  ↩
    reentrant = False)
```

During the build phase, the correct compilation order is computed based on the extensions given:

```
$ waf distclean configure build
'distclean' finished successfully (0.000s)
'configure' finished successfully (0.090s)
Waf: Entering directory `/comp/waf/demos/simple_scenarios/chaining/build'
[1/3] a: uh.in -> build/default/uh.a
[2/3] b: build/default/uh.a -> build/default/uh.b
[3/3] c: build/default/uh.b -> build/default/uh.c
Waf: Leaving directory `/comp/waf/demos/simple_scenarios/chaining/build'
'build' finished successfully (0.034s)
```

8.3 Scanner methods

Because transformation chains rely on implicit transformations, it may be desirable to hide some files from the list of sources. Or, some dependencies may be produced conditionally and may not be known in advance. A *scanner method* is a kind of callback used to find additional dependencies just before the target is generated. For illustration purposes, let us start with an empty project containing three files: the *wscript*, *ch.in* and *ch.dep*

```
$ cd /tmp/smallproject

$ tree
.
|-- ch.dep
|-- ch.in
`-- wscript
```

The build will create a copy of *ch.in* called *ch.out*. Also, *ch.out* must be rebuild whenever *ch.dep* changes. This corresponds more or less to the following Makefile:

```
ch.out: ch.in ch.dep
        cp ch.in ch.out
```

The user script should only contain the following code:

```
top='.'
out='out'
```

```
def configure(conf):
        pass

def build(bld):
        bld(source = 'ch.in')
```

The code below is independent from the user scripts and may be located in a Waf tool.

```
def scan_meth(task): ❶
        node = task.inputs[0]
        dep = node.parent.find_resource(node.name.replace('.in', '.dep')) ❷
        if not dep:
                raise ValueError("Could not find the .dep file for %r" % node)
        return ([dep], []) ❸

import TaskGen
TaskGen.declare_chain(
        name      = 'copy',
        rule      = 'cp ${SRC} ${TGT}',
        ext_in    = '.in',
        ext_out   = '.out',
        reentrant = False,
        scan      = scan_meth, ❹
)
```

❶ The scanner method accepts a task object as input (not a task generator)

❷ Use node methods to locate the dependency (and raise an error if it cannot be found)

❸ Scanner methods return a tuple containing two lists. The first list contains the list of node objects to depend on. The second list contains private data such as debugging information. The results are cached between build calls so the contents must be serializable.

❹ Add the scanner method to chain declaration

The execution trace will be the following:

```
$ echo 1 > ch.in
$ echo 1 > ch.dep ❶

$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/smallproject/out'
[1/1] copy: ch.in -> out/default/ch.out ❷
Waf: Leaving directory `/tmp/smallproject/out'
'build' finished successfully (0.010s)

$ waf
Waf: Entering directory `/tmp/smallproject/out'
Waf: Leaving directory `/tmp/smallproject/out'
'build' finished successfully (0.005s) ❸

$ echo 2 > ch.dep ❹

$ waf
Waf: Entering directory `/tmp/smallproject/out'
[1/1] copy: ch.in -> out/default/ch.out ❺
Waf: Leaving directory `/tmp/smallproject/out'
'build' finished successfully (0.012s)
```

1. Initialize the file contents of *ch.in* and *ch.dep*
2. Execute a first clean build. The file *ch.out* is produced
3. The target *ch.out* is up-to-date because nothing has changed
4. Change the contents of *ch.dep*
5. The dependency has changed, so the target is rebuilt

Here are a few important points about scanner methods:

1. they are executed only when the target is not up-to-date.
2. they may not modify the *task* object or the contents of the configuration set *task.env*
3. they are executed in a single main thread to avoid concurrency issues
4. the results of the scanner (tuple of two lists) are re-used between build executions (and it is possible to access programatically those results)
5. the make-like rules also accept a *scan* argument (scanner methods are bound to the task rather than the task generators)
6. they are used by Waf internally for c/c++ support, to add dependencies dynamically on the header files (*.c* $\rightarrow$ *.h*)

8.4 Extension callbacks

In the chain declaration from the previous sections, the attribute *reentrant* was described to control if the generated files are to be processed or not. There are cases however where one of the two generated files must be declared (because it will be used as a dependency) but where it cannot be considered as a source file in itself (like a header in c/c++). Now consider the following two chains (*uh.in* $\rightarrow$ *uh.a1* + *uh.a2*) and (*uh.a1* $\rightarrow$ *uh.b*) in the following example:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        obj = bld(source='uh.in')

import TaskGen
TaskGen.declare_chain(
        name      = 'a',
        action    = 'cp ${SRC} ${TGT}',
        ext_in    = '.in',
        ext_out   = ['.a1', '.a2'],
        reentrant = True,
)

TaskGen.declare_chain(
        name      = 'b',
        action    = 'cp ${SRC} ${TGT}',
        ext_in    = '.a1',
        ext_out   = '.b',
        reentrant = False,
)
```

The following error message will be produced:

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/smallproject'
Waf: Leaving directory `/tmp/smallproject'
Cannot guess how to process bld:///tmp/smallproject/uh.a2 (got mappings ['.a1', '.in'] in
   class TaskGen.task_gen) -> try conf.check_tool(..)?
```

The error message indicates that there is no way to process *uh.a2*. Only files of extension *.a1* or *.in* can be processed. Internally, extension names are bound to callback methods. The error is raised because no such method could be found, and here is how to register an extension callback globally:

```
@TaskGen.extension('.a2')
def foo(*k, **kw):
        pass
```

To register an extension callback locally, a reference to the task generator object must be kept:

```
def build(bld):
        obj = bld(source='uh.in')
        def callback(*k, **kw):
                pass
        obj.mappings['.a2'] = callback
```

The exact method signature and typical usage for the extension callbacks is the following:

```
import TaskGen
@TaskGen.extension(".a", ".b") ❶
def my_callback(task_gen_object❷, node❸):
        task_gen_object.create_task(
                task_name, ❹
                node,  ❺
                output_nodes) ❻
```

❶ Comma-separated list of extensions (strings)

❷ Task generator instance holding the data

❸ Instance of Node, representing a file (either source or build)

❹ The first argument to create a task is the name of the task class

❺ The second argument is the input node (or a list of nodes for several inputs)

❻ The last parameter is the output node (or a list of nodes for several outputs)

The creation of new task classes will be described in the next section.

8.5 Task class declaration

Waf tasks are instances of the class Task.TaskBase. Yet, the base class contains the real minimum, and the immediate subclass *Task.Task'is usually chosen in user scripts. We will now start over with a simple project containing only one project 'wscript* file and and example file named *ah.in*. A task class will be added.

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        bld(source='uh.in')

import Task, TaskGen

@TaskGen.extension('.in')
def process(self, node):
        tsk = self.create_task('abcd') ❶
        print(tsk.__class__)

class abcd(Task.Task): ❷
        def run(self): ❸
                print('executing...')
                return 0 ❹
```

❶ Create a new instance of *abcd*. The method *create_task* is a shortcut to make certain the task will keep a reference on its task generator.

❷ Inherit the class Task located in the module Task.py

❸ The method run is called when the task is executed

❹ The task return status must be an integer, which is zero to indicate success. The tasks that have failed will be executed on subsequent builds

The output of the build execution will be the following:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/simpleproject/out'
<class 'wscript_main.abcd'>
[1/1] abcd:
executing...
Waf: Leaving directory `/tmp/simpleproject/out'
'build' finished successfully (0.005s)
```

Although it is possible to write down task classes in plain python, two functions (factories) are provided to simplify the work, for example:

```
Task.simple_task_type( ❶
        'xsubpp', ❷
        rule    = '${PERL} ${XSUBPP} ${SRC} > ${TGT}', ❸
        color   = 'BLUE', ❹
        before  = 'cc') ❺

def build_it(task):
        return 0

Task.task_type_from_func(❻
        'sometask', ❼
        func    = build_it, ❽
        vars    = ['SRT'],
        color   = 'RED',
        ext_in  = '.in',
        ext_out = '.out') ❾
```

❶ Create a new task class executing a rule string

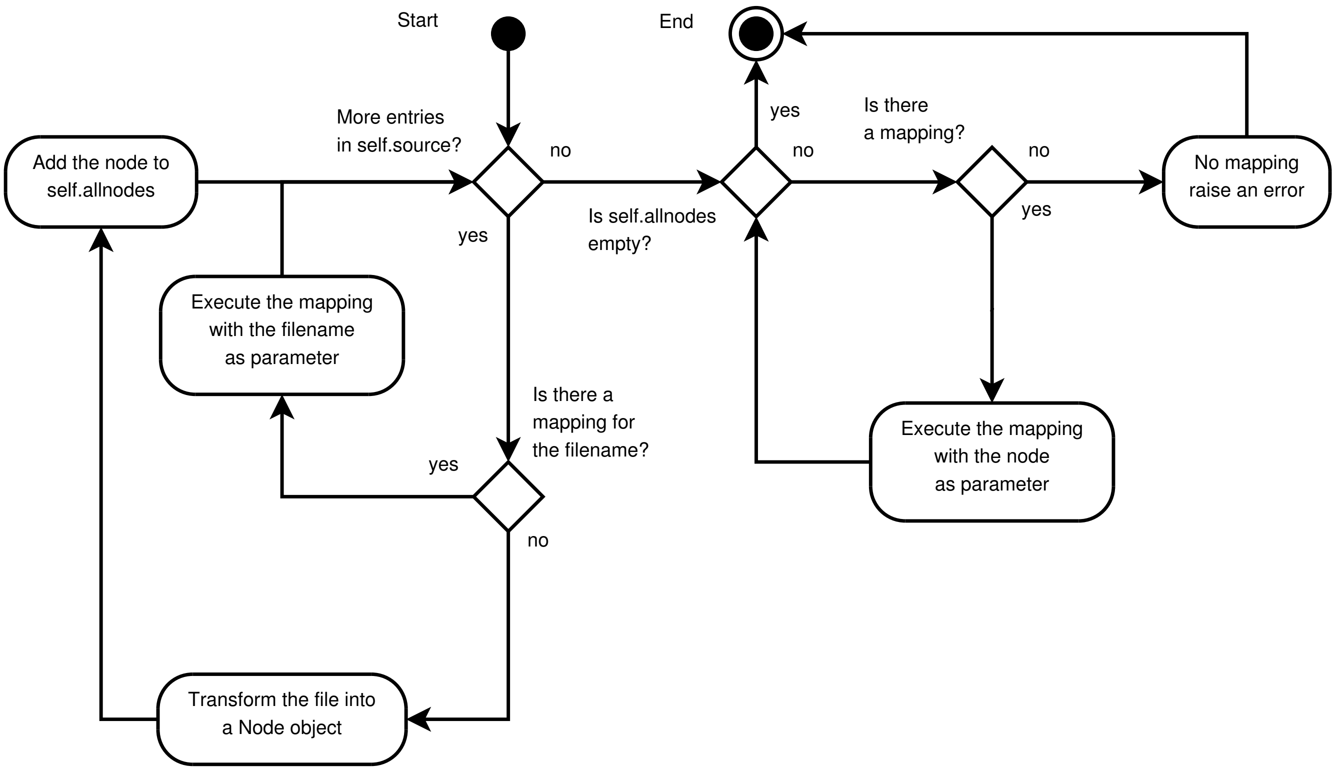
2. Task class name
3. Rule to execute during the build
4. Color for the output during the execution
5. Execute the task instance before any instance of task classes named *cc*. The opposite of *before* is *after*
6. Create a new task class from a custom python function. The *vars* attribute represents additional configuration set variables to use as dependencies
7. Task class name
8. Function to use
9. In this context, the extension names are meant to be used for computing the execution order with other tasks, without naming the other task classes explicitly

Note that most attributes are common between the two function factories. More usage examples may be found in most Waf tools.

8.6 Source attribute processing

The first step in processing the source file attribute is to convert all file names into Nodes. Special methods may be mapped to intercept names by the exact file name entry (no extension). The Node objects are then added to the task generator attribute *allnodes*.

The list of nodes is then consumed by regular extension mappings. Extension methods may re-inject the output nodes for further processing by appending them to the the attribute *allnodes* (hence the name re-entrant provided in declare_chain).



Chapter 9

General purpose task generators

So far, various task generators uses have been demonstrated. This chapter provides a detailed description of task generator structure and usage.

9.1 Task generator definition

The chapter on make-like rules illustrated how the attribute *rule* is processed. Then the chapter on name and extension-based file processing illustrated how the attribute *source* is processed (in the absence of the rule attribute). To process *any attribute*, the following properties should hold:

1. Attributes should be processed only when the task generator is set to generate the tasks (lazy processing)
2. There is no list of authorized attributes (task generators may be extended by user scripts)
3. Attribute processing should be controlable on a task generator instance basis (special rules for particular task generators)
4. The extensions should be split into independent files (low coupling between the Waf tools)

Implementing such a system is a difficult problem which lead to the creation of very different designs:

1. *A hierarchy of task generator subclasses* It was abandoned due to the high coupling between the Waf tools: the C tools required knowledge from the D tool for building hybrid applications
2. *Method decoration (creating linked lists of method calls)* Replacing or disabling a method safely was no longer possible (addition-only), so this system disappeared quickly
3. *Flat method and execution constraint declaration* The concept is close to aspect-oriented programming and may scare programmers.

So far, the third design proved to be the most flexible and was kept. Here is how to define a task generator method:

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        v = bld(myattr='Hello, world!')
        v.myattr = 'Hello, world!' ❶
        v.myMethod() ❷
```

```
import TaskGen

@TaskGen.taskgen ❸
def myMethod(tgen): ❹
        print(getattr(self, 'myattr', None)) ❺
```

❶ Attributes may be set by arguments or by accessing the object. It is set two times in this example.

❷ Call the task generator method explicitly

❸ Use a python decorator

❹ Task generator methods have a unique argument representing the current instance

❺ Process the attribute *myattr* when present (the case in the example)

The output from the build will be the following:

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/simpleproject/out'
hello world
Waf: Leaving directory `/tmp/simpleproject/out'
'build' finished successfully (0.003s)
```

Note
The method could be bound by using *setattr* directly, like for binding any new method on a python class.

9.2 Executing the method during the build

So far, the task generator methods defined are only executed through explicit calls. Another decorator is necessary to have a task generator executed during the build phase automatically. Here is the updated example:

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        bld(myattr='Hello, world!')

import TaskGen

@TaskGen.taskgen ❶
@TaskGen.feature('*') ❷
def methodName(self):
        print(getattr(self, 'myattr', None))
```

❶ Bind a method to the task generator class (optional when other methods such as *TaskGen.feature* are used)

❷ Bind the method to the symbol *myfeature*

The execution results will be the following:

```
$ waf distclean configure build --zones=task_gen ❶
'distclean' finished successfully (0.004s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/simpleproject/out'
23:03:44 task_gen posting objects (normal)
23:03:44 task_gen posting >task_gen '' of type task_gen defined in dir:///tmp/simpleproject ↩
    > 139657958706768 ❷
23:03:44 task_gen -> exec_rule (139657958706768) ❸
23:03:44 task_gen -> apply_core (139657958706768) ❹
23:03:44 task_gen -> methodName (139657958706768) ❺
Hello, world!
23:03:44 task_gen posted ❻
Waf: Leaving directory `/tmp/simpleproject/out'
23:03:44 task_gen posting objects (normal)
'build' finished successfully (0.004s)
```

❶ The debugging zone *task_gen* is used to display the task generator methods being executed

❷ Display which task generator is being executed

❸ The method *exec_rule* is used to process the *rule*. It is always executed.

❹ The method *apply_core* is used to process the *source* attribute. It is always executed exept if the method *exec_rule* processes a *rule* attribute

❺ Our task generator method is executed, and prints *Hello, world!*

❻ The task generator methods have been executed, the task generator is marked as done (posted)

9.3 Task generator features

So far, the task generator methods we added were declared to be executed by all task generator instances. Limiting the execution to specific task generators requires the use of the *feature* decorator:

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        bld(features='ping')
        bld(features='ping pong')

import TaskGen

@TaskGen.feature('ping')
def ping(self):
        print('ping')

@TaskGen.feature('pong')
def pong(self):
        print('pong')
```

The execution output will be the following:

```
$ waf distclean configure build --zones=task_gen
'distclean' finished successfully (0.003s)
'configure' finished successfully (0.001s)
```

```
Waf: Entering directory `/tmp/simpleproject/out'
16:22:07 task_gen posting objects (normal)
16:22:07 task_gen posting <task_gen '' of type task_gen defined in dir:///tmp/simpleproject ↩
    > 140631018237584
16:22:07 task_gen -> exec_rule (140631018237584)
16:22:07 task_gen -> apply_core (140631018237584)
16:22:07 task_gen -> ping (140631018237584)
ping
16:22:07 task_gen posted
16:22:07 task_gen posting <task_gen '' of type task_gen defined in dir:///tmp/simpleproject ↩
    > 140631018237776
16:22:07 task_gen -> exec_rule (140631018237776)
16:22:07 task_gen -> apply_core (140631018237776)
16:22:07 task_gen -> pong (140631018237776)
pong
16:22:07 task_gen -> ping (140631018237776)
ping
16:22:07 task_gen posted
Waf: Leaving directory `/tmp/simpleproject/out'
16:22:07 task_gen posting objects (normal)
'build' finished successfully (0.005s)
```

Warning
Although the task generator instances are processed in order, the task generator method execution requires a specific declaration for the order of execution. Here, the method *pong_* is executed before the method *ping_*

9.4 Task generator method execution order

To control the execution order, two new decorators need to be added. We will now show a new example with two custom task generator methods *method1* and *method2*, executed in that order:

```
top = '.'
out = 'out'

def configure(conf):
        pass

def build(bld):
        bld(myattr='Hello, world!')

import TaskGen

@TaskGen.feature('*')
@TaskGen.before('apply_core', 'exec_rule')
def method1(self):
        print('method 1 %r' % getattr(self, 'myattr', None))

@TaskGen.feature('*')
@TaskGen.before('apply_core')
@TaskGen.after('method1')
def method2(self):
        print('method 2 %r' % getattr(self, 'myattr', None))
```

The execution output will be the following:

```
 waf distclean configure build --zones=task_gen
```

```
'distclean' finished successfully (0.003s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/simpleproject/out'
15:54:02 task_gen posting objects (normal)
15:54:02 task_gen posting <task_gen of type task_gen defined in dir:///tmp/simpleproject>  ↩
    139808568487632
15:54:02 task_gen -> method1 (139808568487632)
method 1 'Hello, world!'
15:54:02 task_gen -> exec_rule (139808568487632)
15:54:02 task_gen -> method2 (139808568487632)
method 2 'Hello, world!'
15:54:02 task_gen -> apply_core (139808568487632)
15:54:02 task_gen posted
Waf: Leaving directory `/tmp/simpleproject/out'
15:54:02 task_gen posting objects (normal)
'build' finished successfully (0.005s)
```

9.5 Adding or removing a method for execution

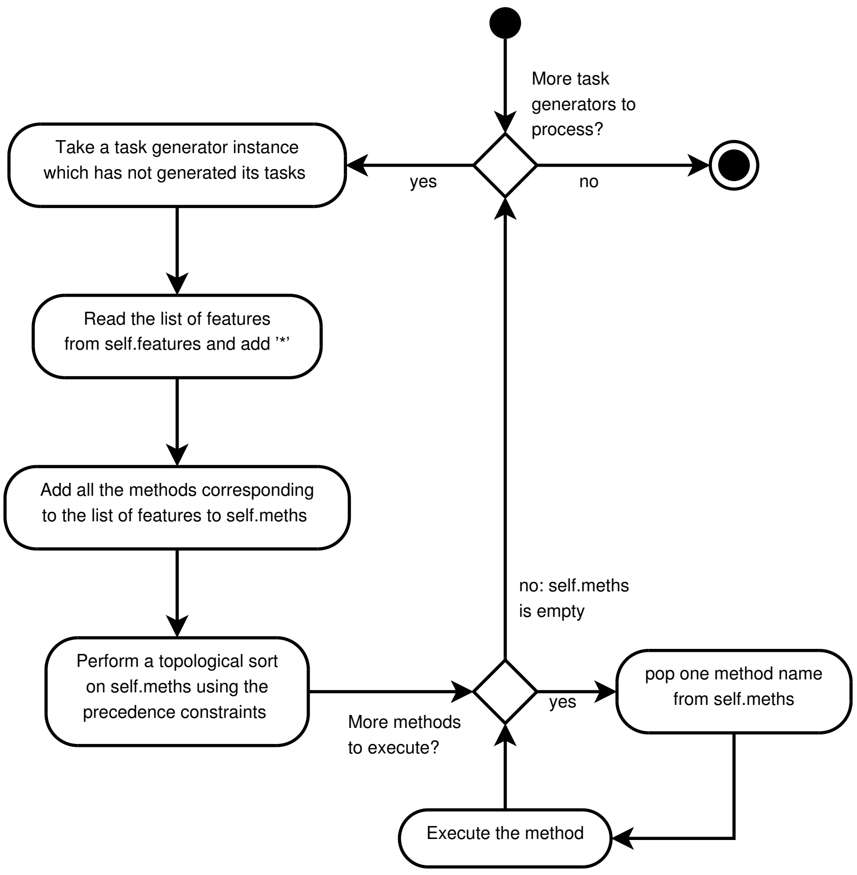
The order constraints on the methods (after/before), are used to sort the list of methods in the attribute *meths*. The sorting is performed once, and the list is consumed as methods are executed. Though no new feature may be added once the first method is executed, new methods may be added dynamically in self.meths. Here is how to create an infinite loop by adding the same method at the end:

```
from TaskGen import feature

@feature('*')
def infinite_loop(self):
        self.meths.append('infinite_loop')
```

Likewise, methods may be removed from the list of methods to execute:

```
from TaskGen import feature

@feature('*')
@before('apply_core')
def remove_apply_core(self):
        self.meths.remove('apply_core')
```

The task generator method workflow is represented in the following illustration:


More task generators to process?
yes
no
Take a task generator instance which has not generated its tasks
Read the list of features from self.features and add '*'
Add all the methods corresponding to the list of features to self.meths
Perform a topological sort on self.meths using the precedence constraints
no: self.meths is empty
More methods to execute?
yes
pop one method name from self.meths
Execute the method

9.6 Expressing abstract dependencies between task generators

We will now illustrate how task generator methods can be used to express abstract dependencies between task generator objects. Here is a new project file located under */tmp/targets/*:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(bld):
        bld(rule='echo A', always=True, name='A')
        bld(rule='echo B', always=True, name='B')
```

By executing *waf --targets=B*, only the task generator *B* will create its tasks, and the output will be the following:

```
$ waf distclean configure build --targets=B
'distclean' finished successfully (0.000s)
'configure' finished successfully (0.042s)
Waf: Entering directory `/tmp/targets/build'
[1/1] B:
B
Waf: Leaving directory `/tmp/targets/build'
'build' finished successfully (0.032s)
```

Here is a way to ensure that the task generator *A* has created its tasks when *B* does:

```
top = '.'
out = 'build'
```

```
def configure(conf):
    pass

def build(bld):
    bld(rule='echo A', always=True, name='A')
    bld(rule='echo B', always=True, name='B', depends_on='A')

from TaskGen import feature, before
@feature('*') ❶
@before('apply_rule')
def post_the_other(self):
    deps = getattr(self, 'depends_on', []) ❷
    for name in self.to_list(deps):
        other = self.bld.name_to_obj(name, self.env) ❸
        print('other task generator tasks (before) %s' % other.tasks)
        other.post() ❹
        print('other task generator tasks (after) %s' % other.tasks)
```

❶ This method will be executed for all task generators, before the attribute `rule` is processed

❷ Try to process the attribute `depends_on`, if present

❸ Obtain the task generator by name, and for the same variant

❹ Force the other task generator to create its tasks

The output will be:

```
$ waf distclean configure build --targets=B
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.001s)
Waf: Entering directory `/tmp/targets/build'
other task generator tasks (before) [] ❶
other task generator tasks (after) [ ❷
        {task: A  -> }]
[1/2] B:
B
[2/2] A: ❸
A
Waf: Leaving directory `/tmp/targets/build'
'build' finished successfully (0.014s)
```

❶ The other task generator has not created any task yet

❷ A task generator creates all its tasks by calling its method `post()`

❸ Although `--targets=B` was requested, the task from target *A* was created and executed too

In practice, the dependencies will often re-use the task objects created by the other task generator: node, configuration set, etc. This is used by the uselib system (see the next chapter on c/c++ builds).

Chapter 10

The Task system

Task objects are created by task generators. They represent atomic transformations performed in the build phase.

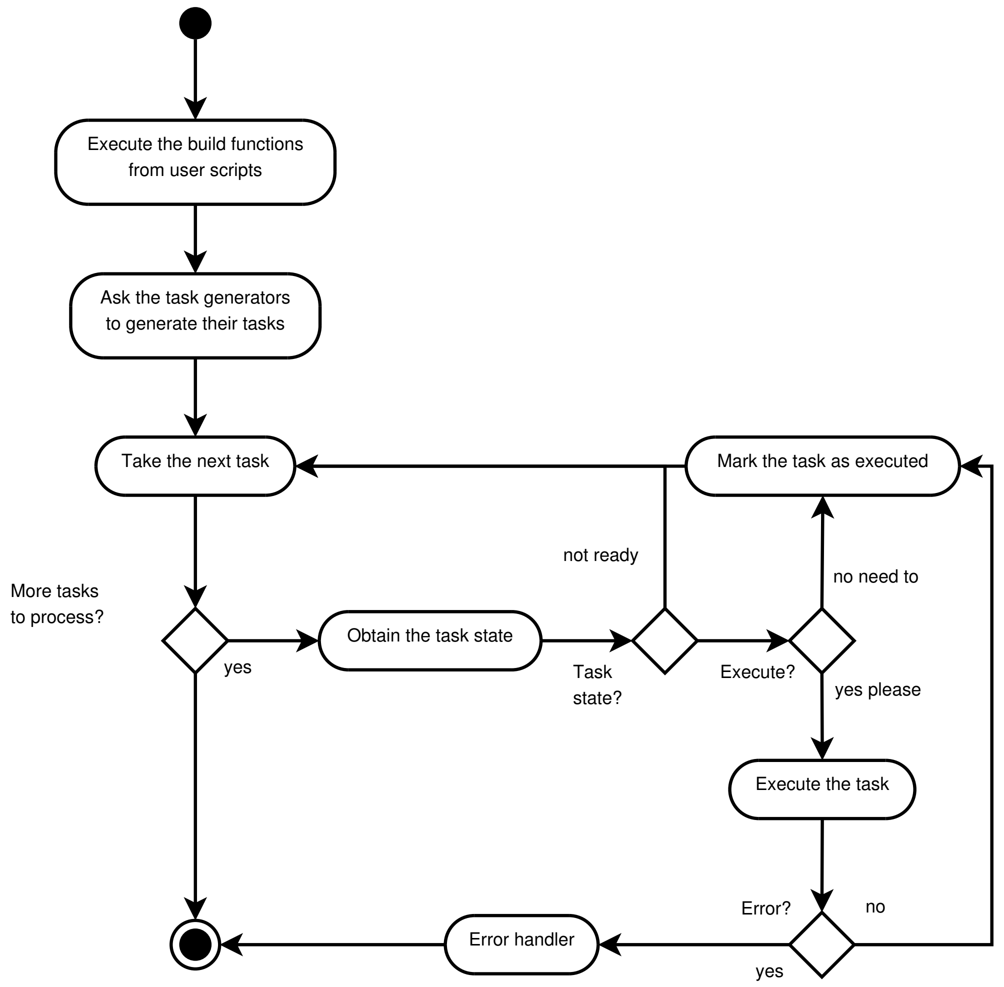
10.1 Task creation and execution

Creating all tasks by hand is a tedious process that the task generators may automate. Before starting the build, Waf asks each task generator to produce the corresponding tasks. If Waf is launched from a sub folder inside the source directory, it will try to avoid the creation of the tasks that are not relevant for that particular sub folder (optimization).

Here are a few motives for using the task generators to create the tasks instead of creating them manually:

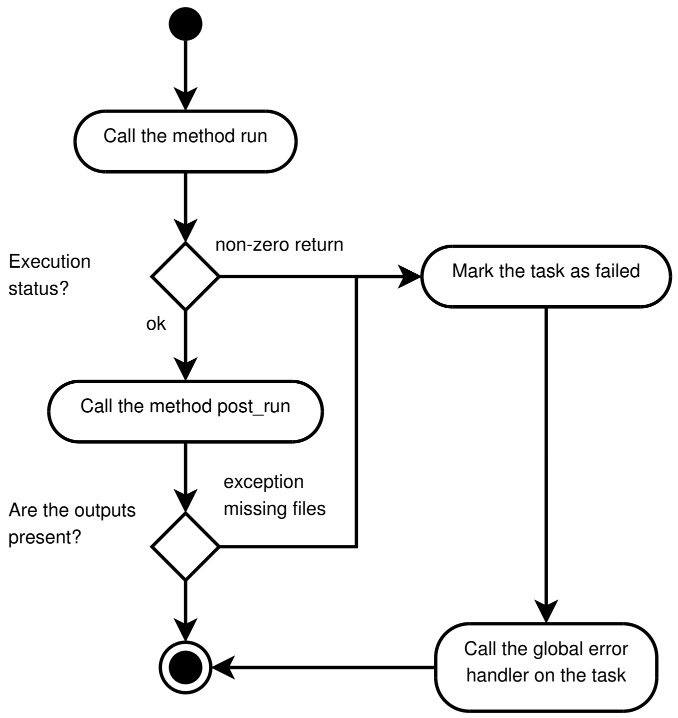
1. Lazy creation: building only a part of the project is a common scenario
2. Filesystem access: the filesystem requirements such as new folders will be created ahead of time
3. Concurrency issues: tasks are executed in parallel and may lead to complex bugs if initialized improperly

A summary can be found on the following diagram:



10.2 Task execution

Executing a task consists in calling the method *run* on that task, and setting the task execution state. The following diagram is a summary of the process:



The method *post_run* can be used to check if the files have been produced, it must throw an OSError if the task has not completed properly.

10.3 Task execution in parallel

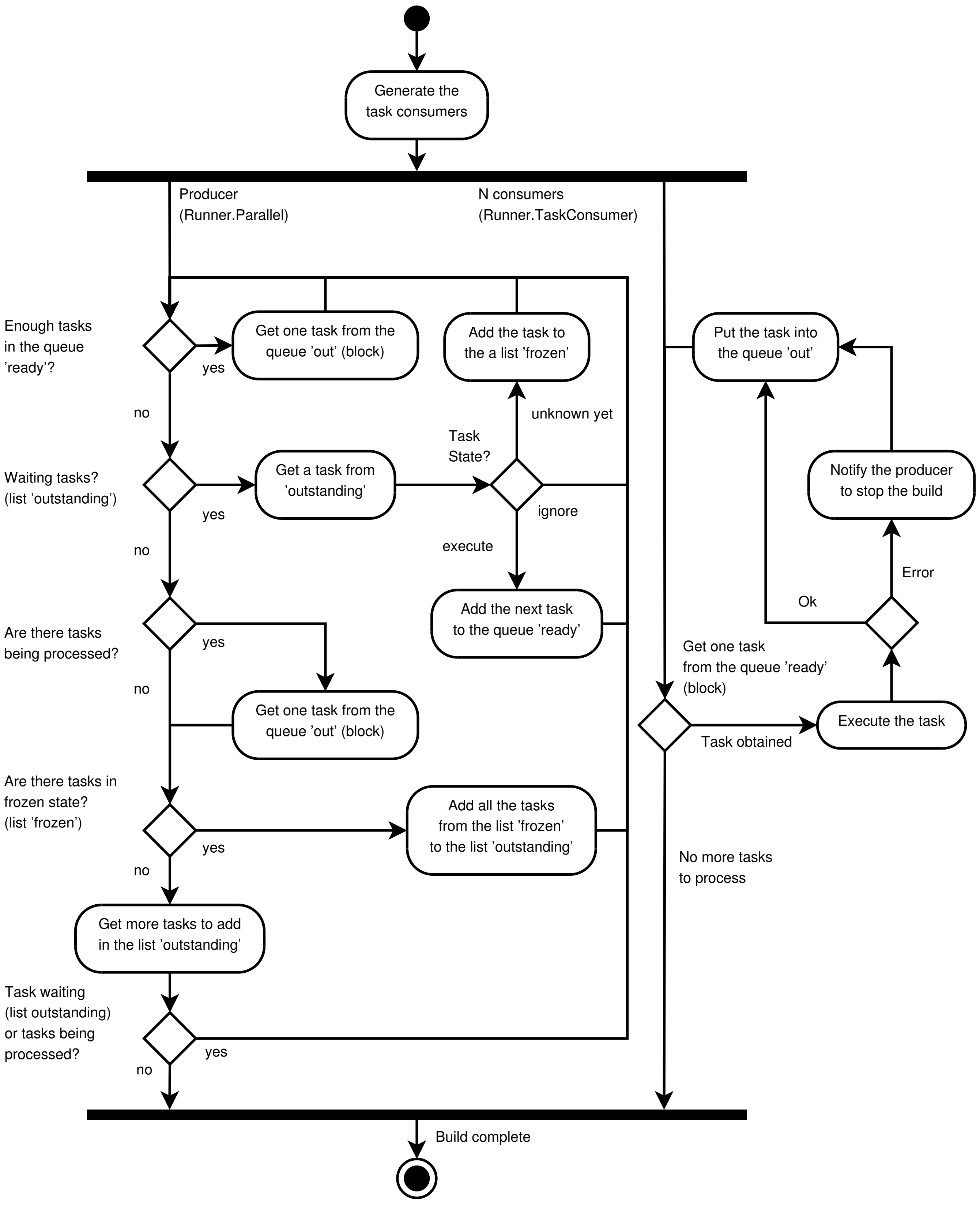
Tasks may be executed in parallel to take advantage of the hardware (multi-core) or the environment (distributed builds). By default Waf does not execute immediately the tasks that are ready. Instead, tasks are added to a queue which is consumed by threads. Waf detects the number of installed processors. For uni-processor only one task is executed at a time, for dual-processors two tasks are executed at a time, and so on. To disable task parallelization, use the option *-j1*. To enhance parallelization, use the option *-j* with the amount of consumers:

```
$ waf -j3
```

By default, Waf does not allow consumer threads to access the tasks directly:

1. There is little need for parallelizing the computation of the next task to execute, choosing the next task is fast enough
2. The thread issues are limited to a very small section of the code
3. The producer-consumer scheme prevents *busy waiting* for the next task
4. A simple global error handler can be used for processing the errors and to decide to stop the build

The following illustrates the relationship producer-consumer performed for the builds:


Generate the task consumers
Producer (Runner.Parallel)
N consumers (Runner.TaskConsumer)
Enough tasks in the queue 'ready'?
yes
Get one task from the queue 'out' (block)
Add the task to the a list 'frozen'
Put the task into the queue 'out'
no
unknown yet
Task State?
Waiting tasks? (list 'outstanding')
Get a task from 'outstanding'
Notify the producer to stop the build
yes
ignore
no
execute
Error
Ok
Add the next task to the queue 'ready'
Are there tasks being processed?
yes
Get one task from the queue 'ready' (block)
no
Get one task from the queue 'out' (block)
Task obtained
Execute the task
Are there tasks in frozen state? (list 'frozen')
Add all the tasks from the list 'frozen' to the list 'outstanding'
yes
no
No more tasks to process
Get more tasks to add in the list 'outstanding'
Task waiting (list outstanding) or tasks being processed?
yes
no
Build complete

10.4 Task execution order

Running tasks in parallel is a simple problem, but in practice it is more complicated: . Dependencies can be discovered during the build (dynamic task creation) . New ordering constraints can be discovered after files are compiled . The amount of tasks and ordering constraints (graph size) can be huge and performance may be a problem

To make the problem more simple, it is divided by the different concerns, and the ordering constraints can be given on three different levels:

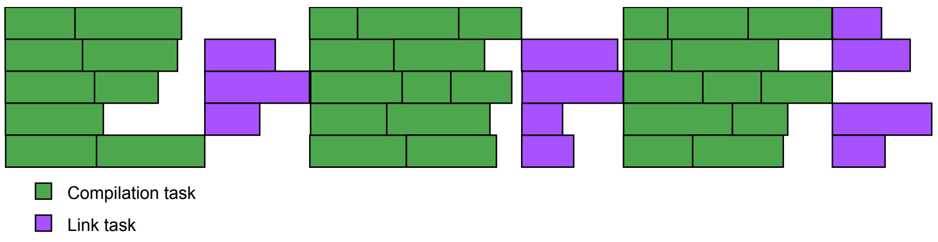
1. groups of tasks may run only after another group of tasks has finished to run, this represents a strict sequential order between groups of tasks, for example a compiler is produced and used to compile the tasks in the next group
2. task types to indicate the instance will run after other task type instances, for example linking object files may only occur after compiling the source files
3. specific constraints for task instances that can only run after a few other task instances

10.4.1 Task groups

In some circumstances it is necessary to build a compiler and all its dependencies before using it for executing some other tasks (bootstrapping). The following demonstrates how declare groups of tasks to be executed after other groups of tasks:

```
def build(bld):
        bld(features='cc cprogram', source='main.c', target='mycompiler')
        bld.add_group()
        bld(features='cc cprogram', source='user.c', target='someotherapp')
```

The effect of task groups when running tasks in parallel is illustrated by the following diagram. Three groups of tasks have been added, and the execution of the next group only starts when the execution of the tasks in the previous group is complete.



Tasks in three groups executed with −j5

It is possible to create groups at any point in the scripts, and to add the task generators to any group previously created. Adding groups for specific folders or scripts enables a behaviour similar to projects organized in recursive Makefiles.

```
def build(bld):

        bld.add_group('test1')
        bld.add_group('test2')
        bld.add_group('test3')
        bld.add_group('test4')

        print('adding task generators')

        bld.set_group('test3')
        bld(features='cxx cprogram', source='main3.c', target='g3')

        bld.set_group('test1')
        bld(features='cxx cprogram', source='main1.c', target='g1')

        bld.set_group('test2')
        obj2 = bld(features='cxx cprogram', source='main2.c', target='g2')

        bld.set_group('test4')
        obj2.clone('debug')
```

Because task groups prevent parallelization, they reduce performance. On the other hand, they make projects more structured and improve the maintainance.

10.4.2 Precedence constraints

The attributes *before* and *after* are used to declare ordering constraints between tasks:

```
import Task
class task_test_a(Task.TaskBase):
        before = 'task_test_b'
class task_test_b(Task.TaskBase):
        after = 'task_test_a'
```

Another way to declare precedence constraints is to declare a file extension production, for example:

```
import Task
class task_test_a(Task.TaskBase):
        ext_in = '.c'
class task_test_b(Task.TaskBase):
        ext_out = '.c'
```

The *extensions* ext_in and ext_out have to match to add a valid precedence constraint, but they are only symbols in this context. They do not mean the tasks actually have to produce files of that type.

10.4.3 Precedence constraints on task instances

The method *set_run_after* is used to declare ordering constraints between tasks:

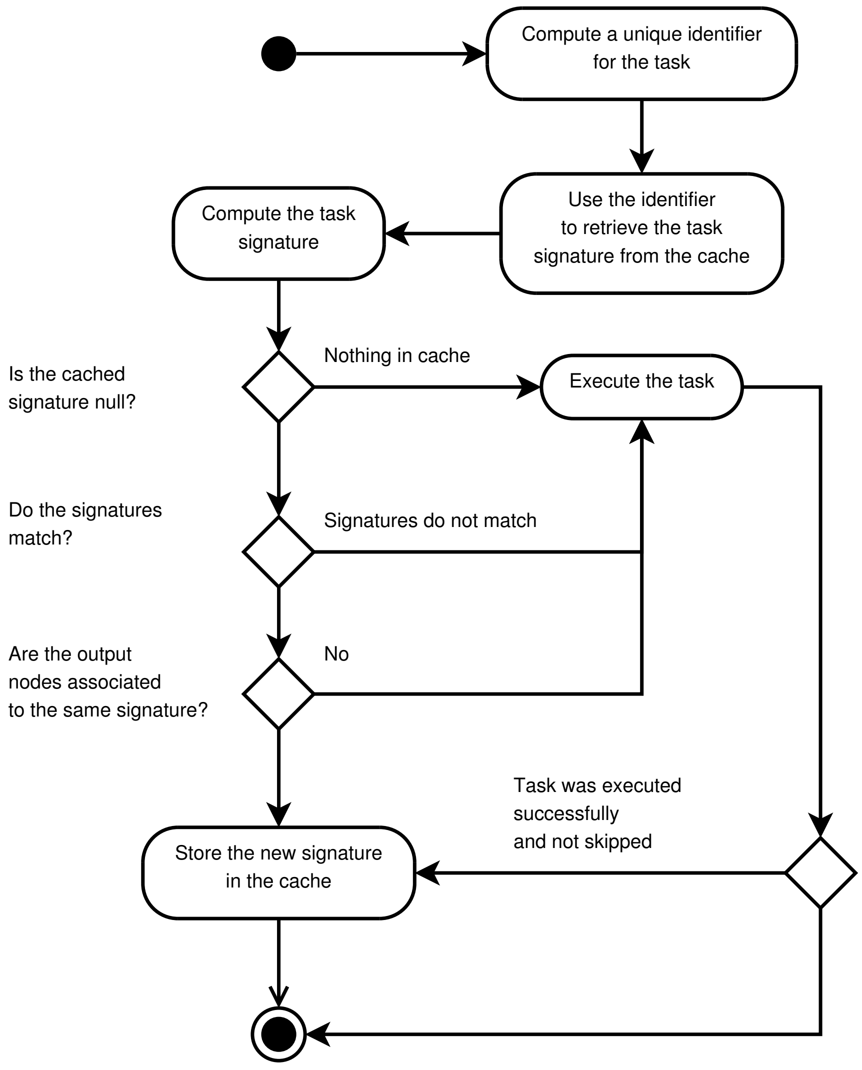
```
task1.set_run_after(task2)
```

unlike the previous constraints, it is used on the instances of class *Task* which is a subclass of class *TaskBase*

10.5 Executing tasks only when something changes

The direct instances of TaskBase are quite limited and do not track the changes to the source files. The class *Task* provides the necessary features for the most common builds in which source files are used to produce target files. The idea is to create a unique signature for tasks, and to represent the dependencies on files or other tasks by including them in the signature. A hashing function is used for computing the signature, by default it is md5.

The following diagram illustrates the task processing including the signature, it is only valid for Task instance (not TaskBase instances):


Compute a unique identifier for the task
Use the identifier to retrieve the task signature from the cache
Compute the task signature
Is the cached signature null?
Nothing in cache
Execute the task
Do the signatures match?
Signatures do not match
Are the output nodes associated to the same signature?
No
Task was executed successfully and not skipped
Store the new signature in the cache

The signature computation uses the following data:

1. explicit dependencies: input files and dependencies set explicitly using task.deps_man or bld.depends_on
2. implicit dependencies: dependencies searched by the task itself (like source files included from other source files).
3. parameters: compilation flags and command-line parameters.

Here is an example illustrating the different kinds of dependencies:

```
import Task
class task_demo(Task.Task):
        vars = ['CXXFLAGS', 'LINKFLAGS'] ❶
        def scan(self): ❷
                return [[self.inputs[0].parent.find_resource('.svn/entries')], []]

task = task_demo()
task.inputs = [bld.path.find_resource('test.cxx')] ❸
task.deps_man = [bld.path.find_resource('wscript')] ❹

bld.add_manual_dependency('main.c', 'an arbitrary string value') ❺
bld.add_manual_dependency(
                bld.path.find_or_declare('test_c_program'),
                bld.path.find_resource('bbb')) ❻
```

1. Environment variable dependencies (compilation flags)
2. Implicit dependencies: a method returns a list containing the list of additional nodes to take into account, and the list of the files that could not be found (cache)
3. Explicit dependencies as input files (nodes)
4. Explicit dependencies as manual dependencies
5. Manual dependencies on source files, the second parameter can be a string, a node object or a function returning a string
6. Manual dependencies with nodes, the first node represents a target (which may or may not exist in the build), and the second parameter represents a file in the source directory.

Chapter 11

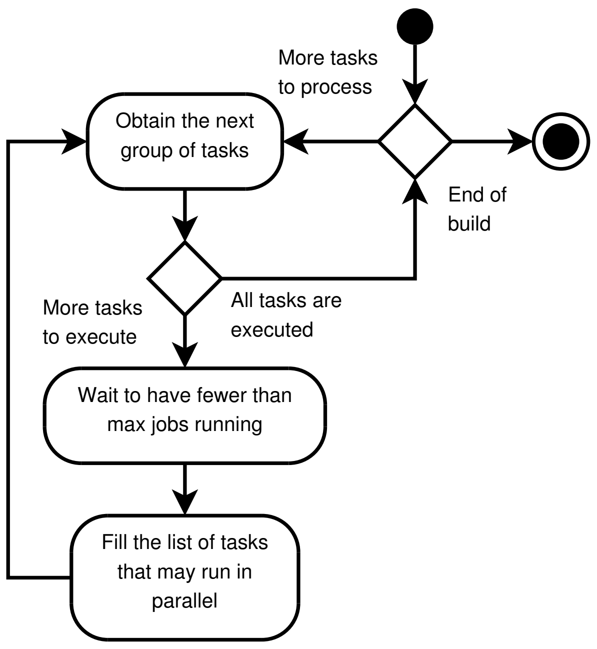
The scheduler for executing the tasks

Tasks are executed in parallel during the build phase, yet with a few restrictions.

11.1 The task execution model

Task dependencies and task ordering specify the exact order in which tasks must be executed. When tasks are executed in parallel, different algorithms may be used to improve the compilation times. For example, tasks that are known to last longer may be launched first. Linking tasks that use a lot of ram (in the context of c++ applications) may be launched alone to avoid disk thrashing by saving RAM.

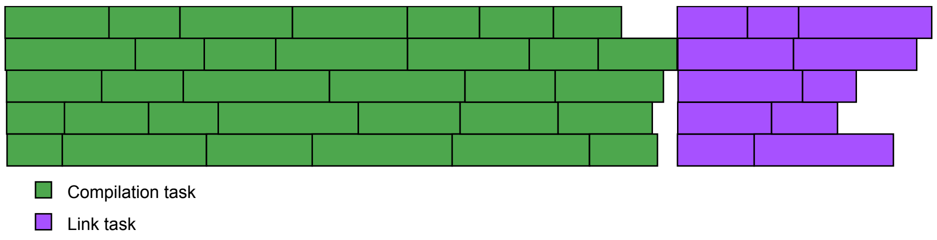
To make this possible, the task execution is organized in the following manner:


More tasks to process
Obtain the next group of tasks
End of build
More tasks to execute
All tasks are executed
Wait to have fewer than max jobs running
Fill the list of tasks that may run in parallel

11.2 Job control

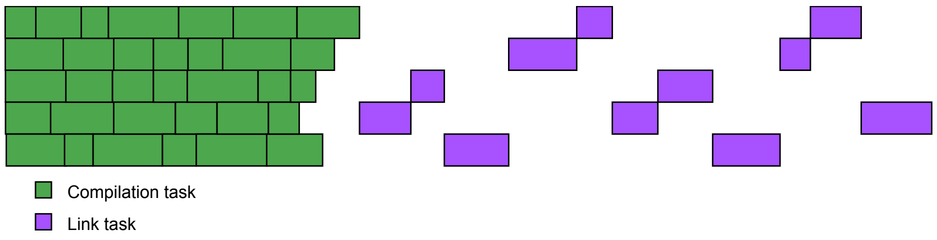
Job control is related to the parallelization algorithms used for launching the tasks. While the aim of parallelization is to maximize the amount of tasks executed in parallel, different algorithms may be used

In the NORMAL ordering, task groups are created, and a topological sort is performed on the task class types. The overall performance penalty for complete builds is usually small, like a few seconds on builds during minutes.



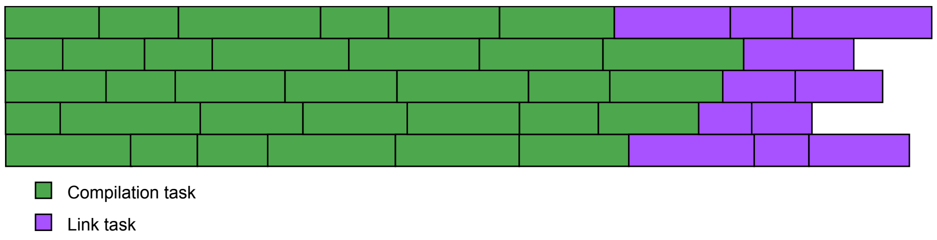
Tasks executed with −j5 and NORMAL

In the JOBCONTROL ordering, groups are created in advance, and a flag indicates the maximum amount of jobs to be used when the consumer threads execute the tasks. This prevents parallelization of tasks which use a lot of resources. For example, linking c++ object files uses a lot of RAM.



Tasks executed with −j5 and JOBCONTROL

In the MAXPARALLEL ordering, Each task holds a list of tasks it must run after (there is only one list of tasks waiting to be executed). Though this version parallelizes tasks very well, it consumes more memory and processing. In practice, Waf may last 20% more on builds when all tasks are up-to-date.



Tasks executed with −j5 and MAXPARALLEL

Warning
Because most task classes use ordering constraints, the maximum parallelization can only be achieved if the constraints between task classes are relaxed, and if all task instances know their predecessors. In the example graph, this was achieved by removing the ordering constraints between the compilation tasks classes and the link tasks classes.

```
import Task
Task.TaskBase.classes['cxx'].ext_out = [] ❶

import Runner
old_refill = Runner.Parallel.refill_task_list
def refill_task_list(self): ❷
    old_refill(self)
    lst = self.outstanding
    if lst:
        for x in lst: ❸
            for y in lst:
                for i in x.inputs: ❹
                    for j in y.outputs:
                        if i.id == j.id:
                            x.set_run_after(y) ❺
Runner.Parallel.refill_task_list = refill_task_list
```

❶ relax the constraints between cxx and cxx_link (in the build section)

❷ override the definition of Runner.Parallel.refill_task_list

❸ consider all task instances

❹ infer the task orderings from input and output nodes

❺ apply the constraint order

From this, we can immediately notice the following:

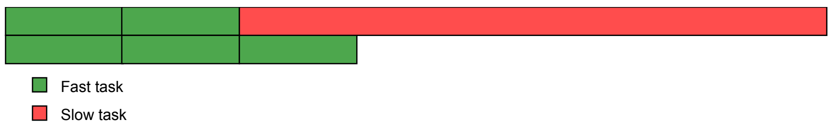
1. An assumption is made that all tasks have input and output nodes, and that ordering constraints can be deduced from them
2. Deducing the constraints from the input and output nodes exhibits a n^2 behaviour

Note
In practice, the NORMAL algorithm should be used whenever possible, and the MAXPARALLEL should be used if substantial gains are expected and if the ordering is specified between all tasks. The JOBCONTROL system may be useful for tasks that consume a vast amount of resources.

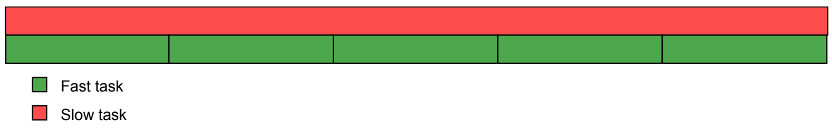
11.3 Weak task order constraints

Tasks that are known to take a lot of time may be launched first to improve the build times. The general problem of finding an optimal order for launching tasks in parallel and with constraints is called Job Shop. In practice this problem can often be reduced to a critical path problem (approximation).

The following pictures illustrate the difference in scheduling a build with different independent tasks, in which a slow task is clearly identified, and launched first:



Tasks with different durations executed with −j2



Tasks with different durations executed with −j2 (slowest first)

Waf provides a function for reordering the tasks before they are launched in the module Runner, the default reordering may be changed by dynamic method replacement in Python:

```
import Runner
def get_next(self):
        # reorder the task list by a random function
        self.outstanding.sort()
        # return the next task
        return self.outstanding.pop()
Runner.Parallel.get_next = get_next
```

If the reordering is not to be performed each time a task is retrieved, the list of task may be reordered when the next group is retrieved:

```
import Runner
old_refill = Runner.Parallel.refill_task_list
def refill_task_list(self):
        old_refill(self)
        self.outstanding.sort()
Runner.Parallel.refill_task_list = refill_task_list
```

It is possible to measure the task execution times by intercepting the function calls. The task execution times may be re-used for optimizing the schedule for subsequent builds:

```
import time
import Task
old_call_run = Task.TaskBase.call_run
def new_call_run(self):
        t1 = time.time()
        ret = old_call_run(self)
        t2 = time.time()
        if not ret: print("execution time %r" % (t2 - t1))
        return ret
Task.TaskBase.call_run = new_call_run
```

Chapter 12

Build context and nodes

Node objects are only visible through a few object (bld.path), and are used internally by task objects. Although their usage is somewhat restricted, there are a few useful applications from user scripts.

12.1 Node types

Although nodes are use for representing the filesystem, there are only 3 node types:

1. Source nodes represent source files present on the file system. The signature for a source node is the hash of the file contents.
2. Build nodes represent files created under the build directory. While build nodes are not bound to the build variant, the build nodes may have several signatures corresponding to the variants in which the files have been created (the signature is the signature of the task that created the file).
3. Directory nodes represent the folders from the project directory. During the build phase, the folders corresponding to the source directory are automatically created for each variant into the build directory. Directory nodes have no associated signatures.

For example, for a project located in /tmp/nodes, using the rule *source.in* → *target.out* with three variants *default*, *debug* and *release*, the filesystem contents will be:

```
$ tree
.
|-- build
|   |-- c4che
|   |   |-- build.config.py
|   |   |-- debug.cache.py
|   |   |-- default.cache.py
|   |   `-- release.cache.py
|   |-- config.log
|   |-- debug
|   |   `-- target.out
|   |-- default
|   `-- release
|       `-- target.out
|-- source.in
`-- wscript
```

And the filesystem representation will consist in only three nodes:

```
dir:///tmp/nodes ❶
src:///tmp/nodes/source.in ❷
bld:///tmp/nodes/target.out ❸
```

❶ The project directory. Directory nodes are represented by *dir://*

❷ The source node for *source.in*. Source nodes are represented by *src://*

❸ The build node for the outputs *target.out*. Build nodes are represented by *bld://*

There can be only one node of a name for a given directory node. Because of this restriction, it is not possible to copy a file from the source directory to the corresponding build directory (for a given variant):

```
bld(
        rule   = 'cp ${SRC} ${TGT}',
        source = 'test.png',
        target = 'test.png') ❶

bld(
        rule   = 'cp ${SRC} ${SRC[0].parent.abspath(env)}/${SRC[0].name}',
        source = 'test.png') ❷
```

❶ Forbidden

❷ Discouraged, but Ok.

12.2 Build context creation and access

For testing purposes, a function created from a build context instance could create another build context. Therefore, the build context is not a singleton. The task signatures and the dependent nodes are then bound to the build context instance that created them.

Here is how to create a build context inside another build context:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(ctx):

        def do_it():

                import Environment, Build
                bld = Build.BuildContext() ❶
                bld.load_dirs('/tmp', '/tmp/build') ❷
                env = Environment.Environment()
                bld.all_envs['default'] = env ❸
                bld.init_variants() ❹

                bld( ❺
                        rule   = 'echo hi from the nested context!',
                        always = True)

                bld.compile() ❻

        ctx( ❼
                rule   = 'echo hi from the main buildcontext',
                always = True)

        do_it() ❽
```

❶ Create a new build context.

❷ Set the project and the output folders

❸ Set the configuration data set

❹ Create the folder(s) */tmp/build/variant* and initialize the build node signature cache

❺ Declare a task generator

❻ Execute a build

❼ Create a task generator for the main build context

❽ Call the function *do_it* to create immediately a new build context and to execute it

The execution trace is the following:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.002s)
Waf: Entering directory `/tmp/nested/build'
[1/1] echo hi from the nested context!: ❶
hi from the nested context!
[1/1] echo hi from the main buildcontext:
hi from the main buildcontext ❷
Waf: Leaving directory `/tmp/nested/build'
'build' finished successfully (0.018s)
```

❶ The nested build is executed immediately

❷ Trace from the normal build

The task generators, the tasks, and the node objects are all bound to a particular build context. Here is how to access the different objects:

```
top = '.'
out = 'build'

def configure(conf):
    pass

def build(bld):
    print('root %r' % type(bld.root)) ❶
    print('path %r' % bld.path) ❷
    print('path %r' % bld.srcnode) ❸
    print('bld %r' % bld.path.__class__.bld) ❹

    def fun(task):
        print("task %r %r -> %r" % (
            type(task),
            task.generator, ❺
            task.generator.bld) ❻
        )

    obj = bld(rule=fun, always=True, name='foo')
    print('tgen %r -> %r' % (type(obj), obj.bld)) ❼
    print(bld.name_to_obj('foo', bld.env)) ❽
```

1. Filesystem root
2. Current path
3. Project directory (top-level)
4. Access the build context instance from the class
5. Get a reference to the task generator that created the task instance
6. Get the build context corresponding to the task instance
7. The attribute *bld* of a task generator is the build context
8. Obtain a task generator from the build context

The execution trace will be the following:

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
'configure' finished successfully (0.002s)
Waf: Entering directory `/tmp/nested/build'
root <class 'Node.Nodu'>
path dir:///tmp/nested
src  dir:///tmp/nested
bld <Build.BuildContext object at 0x7f5472764490>
tgen <class 'TaskGen.task_gen'> -> <Build.BuildContext object at 0x7f5472764490>
<task_gen 'foo' of type task_gen defined in dir:///tmp/nested>
[1/1] 1:
task <class 'Task.1'> <TaskGen.task_gen object at 0x7f547277b610>
     -> <Build.BuildContext object at 0x7f5472764490>
Waf: Leaving directory `/tmp/nested/build'
'build' finished successfully (0.007s)
```

Note
Tasks created by task generators are somehow private objects. They should not be manipulated directly in the *build* function, but rather by task generator methods.

12.3 Using nodes

12.3.1 Obtaining nodes

Three main Node methods are commonly used for accessing the file system:

1. find_dir: return a directory node or None if the folder cannot be found on the system. By calling this method, the corresponding folders in the build directory will be created for each variant.
2. find_resource: return a source node, or a build node if a build node with the given name exist, or None if no such node exists. Compute and store the node signature (by hashing the file). This method may call *find_dir* internally.
3. find_or_declare: Return a build node. If no corresponding build node exist, it will be created first. This method may call *find_dir* internally.

These methods interact with the filesystem, and may create other nodes such as intermediate folders. It is important to avoid their usage in a context of concurrency (threading). In general, this means avoiding node manipulations in the methods that execute the tasks.

Although nodes have an attribute named *parent*, it is usually better to access the hierarchy by calling *find_dir* with a relative path. For example:

```
def build(bld):
        p = bld.path.parent.find_dir('src') ❶
        p = bld.path.find_dir('../src') ❷
```

❶ Bad, do not use.

❷ Recommended. Separators such as / are converted automatically and does not need to be the os separator.

Warning
Node instances must not be created manually.

12.3.2 The absolute path

The method *abspath* is used to obtain the absolute path for a node. In the following example, three nodes are used:

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(ctx):
        dir = ctx.path ❶
        src = ctx.path.find_resource('wscript')
        bld = ctx.path.find_or_declare('out.out')

        print(src.abspath(ctx.env)) ❷
        print(bld.abspath(ctx.env))
        print(dir.abspath(ctx.env)) ❸
        print(dir.abspath())
```

❶ Directory node, source node and build node

❷ Computing the absolute path for source node or a build node takes a configuration set as parameter

❸ Computing the absolute path for a directory may use a configuration set or not

Here is the execution trace:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
'configure' finished successfully (0.005s)
Waf: Entering directory `/tmp/nested/build'
/tmp/nested/wscript ❶
/tmp/nested/build/default/out.out ❷
/tmp/nested/build/default/ ❸
/tmp/nested ❹
Waf: Leaving directory `/tmp/nested/build'
'build' finished successfully (0.003s)
```

❶ Absolute path for the source node

❷ The absolute path for the build node depends on the variant in use

3 When a configuration set is provided, the absolute path for a directory node is the build directory representation including the variant

4 When no configuration set is provided, the directory node absolute path is the one for the source directory

Note
Several other methods such as *relpath_gen* or *srcpath* are provided. See the api documentation

12.4 Searching for nodes

12.4.1 ant_glob

The Ant-like Node method *ant_glob* is used for finding nodes or files.

```
top = '.'
out = 'build'

def configure(conf):
        pass

def build(ctx):
        print(ctx.path.ant_glob('**/*', src=True, bld=False, dir=False, flat=False))
        print(ctx.path.ant_glob('**/*', src=True, bld=False, dir=False, flat=True))
```

The results will be:

```
$ waf
wscript .lock-wscript
[src:///tmp/ant/wscript, src:///tmp/ant/.lock-wscript]
```

The behaviour of *ant_glob* will be recursive if the expression contains **. It is therefore a good idea to limit the expression to what is strictly necessary.

12.4.2 update_build_dir

By default, *ant_glob* will only find the files for folders that exist in the build directory. If new folders are created manually in the build directory, they must be declared somehow. Here is an example:

```
top = '.'
out = 'build'

def configure(conf):
    conf.find_program('touch', mandatory=True)

def build(bld):
    bld(rule='mkdir -p default/a/b/c/ && touch default/a/b/c/foo.txt', name='blah')

    def printi(bld):
        print(' before update %r' % bld.path.ant_glob('**/*.txt', bld=True))
        bld.path.update_build_dir(bld.env)
        print(' after update  %r' % bld.path.ant_glob('**/*.txt', bld=True))

    bld.add_post_fun(printi)
```

The tree generated during the build will be the following:

```
$ tree
.
|-- build
|   |-- c4che
|   |   |-- build.config.py
|   |   `-- default.cache.py
|   |-- config.log
|   `-- default
|       `-- a
|           `-- b
|               `-- c
|                   `-- foo.txt
`-- wscript
```

The output from the execution will be:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
Checking for program touch               : ok /usr/bin/touch
'configure' finished successfully (0.003s)
Waf: Entering directory `/comp/waf/demos/simple_scenarios/magic_outputs/build'
[1/1] blah:
Waf: Leaving directory `/comp/waf/demos/simple_scenarios/magic_outputs/build'
 before update '' ❶
 after update  'a/b/c/foo.txt' ❷
'build' finished successfully (0.018s)
```

❶ Files created in the build directory by a foreign process are ignored

❷ Read the contents from the build directory

Note
In *update_build_dir*, the parameter *env* is optional. When unset, all variants are considered.

Chapter 13

C and C++ projects

Although Waf is language neutral, it is commonly used for C and C++ projects. This chapter describes the waf tools used for this purpose.

13.1 Common script for C/C++ applications

The c/c++ builds consist in transforming source files into object files, and to assemble the object files at the end. In theory a single programming language should be sufficient for writing any application, but in practice this does not scale:

1. Applications may be divided in dynamic or static libraries
2. Additional files may enter in the link step (libraries, object files)
3. Source files may be generated by other compilers
4. Differnt platforms require different processing rules (manifest files on windows, etc)

The construction of c/c++ applications can be quite complicated, and several measures must be taken to ensure coherent interaction with new compilation rules. The canonical code for a task generator building a c/c++ application is the following:

```
top = '.'
out = 'build'

def set_options(opt):
        opt.tool_options('compiler_cc')

def configure(conf):
        conf.check_tool('compiler_cc') ❶

def build(bld):
        t = bld(
                features     = ['cc', 'cprogram'], ❷
                source       = 'main.c', ❸
                target       = 'appname', ❹
                install_path = '${SOME_PATH}/bin', ❺
                vnum         = '1.2.3', ❻
                includes     = ['.'], ❼
                defines      = ['LINUX=1', 'BIDULE'], ❽
                ccflags      = ['-O2', '-Wall'], ❾
                lib          = ['m'], ❿
                libpath      = ['/usr/lib'],
                linkflags    = ['-g'],
        )
        t.rpath          = ['/usr/lib']
```

1. Load the c/c++ support routines (such as the features below) and try to find a suitable compiler for the build. The support for c++ is loaded by *compiler_cxx*
2. Task generator declaration; each element in the list represent a feature; it is possible to add several languages at once (*ocaml and c++* for example), but the one of *cstaticlib, cshlib or cprogram* must be chosen.
3. List of source, it may be either a python list, or a string containing the file names separated with spaces. This list may contain file names of different extensions to make hybrid applications.
4. Target name, it is concerted to the name of the binary *name.so* or *name.exe* depending on the platform and the features.
5. Installation directory, this is where to install the library or program produced. The *${}* expression is a reference to a variable to be extracted from *tgen.env*. By default it is set to *${PREFIX}/bin* for programs and *${PREFIX}/lib* for libraries. To disable the installation, set it to *None*.
6. Version number for shared libraries. It is not used for programs or static libraries.
7. List of include paths, it may be either a python list, or a string containing the paths separated by spaces. The paths are used for both the command-line and for finding the implicit dependencies (headers). In general, include paths must be relative to the wscript folder and given explicitly.
8. Command-line defines: list of defines to add to the command-line with the *-D* prefix. To reduce the size of the command-line, it is possible to use a configuration header, see the following section for more details.
9. Command-line compilation flags, for the c++ language the attribute is called *cxxflags*
10. Shared libraries may be given directly (use *staticlib* and *staticlibpath* for static libraries)

Additional pararameters may be added from a task generator reference. The next section describes a technique to gather the conditions into the configuration section.

13.2 Include processing

13.2.1 Execution path and flags

Include paths are used by the c/c++ compilers for finding the headers. When one header changes, the files are recompiled automatically. For example on a project having the following structure:

```
$ tree
.
|-- foo.h
|-- src
|   |-- main.c
|   `-- wscript
`-- wscript
```

The file *src/wscript* will contain the following code:

```
def build(bld):
    bld(
        features = 'cc cprogram',
        source   = 'main.c',
        target   = 'myapp',
        includes = '.. .')
```

The command-line (output by `waf -v`) will have the following form:

```
cc -Idefault -I.. -Idefault/src -I../src ../src/main.c -c -o default/src/main_1.o
```

Because commands are executed from the build directory, the folders have been converted to include flags in the following way:

```
.. -> -I..       -Idefault
.  -> -I../src  -Idefault/src
```

There are the important points to remember:

1. The includes are always given relative to the directory containing the wscript file
2. The includes add both the source directory and the corresponding build directory for the task generator variant
3. Commands are executed from the build directory, so the include paths must be converted
4. System include paths should be defined during the configuration and added to CPPPATH variables (uselib)

13.2.2 The Waf preprocessor

Waf uses a preprocessor written in Python for adding the dependencies on the headers. A simple parser looking at #include statements would miss constructs such as:

```
#define mymacro "foo.h"
#include mymacro
```

Using the compiler for finding the dependencies would not work for applications requiring file preprocessing such as Qt. For Qt, special include files having the *.moc* extension must be detected by the build system and produced ahead of time. The c compiler could not parse such files.

```
#include "foo.moc"
```

Since system headers are not tracked by default, the waf preprocessor may miss dependencies written in the following form:

```
#if SOMEMACRO
        /* an include in the project */
        #include "foo.h"
#endif
```

To write portable code and to ease debugging, it is strongly recommended to put all the conditions used within a project into a *config.h* file.

```
def configure(conf):
        conf.check(
                fragment='int main() { return 0; }\n',
                define_name='FOO',
                mandatory=1)
        conf.write_config_header('config.h')
```

For performance reasons, the dependencies are not added onto system headers by default. The following code may be used to enable this behaviour:

```
import preproc
preproc.go_absolute = True
```

13.2.3 Debugging dependencies

The Waf preprocessor contains a specific debugging zone:

```
$ waf --zones=preproc
```

To display the dependencies obtained or missed, use the following:

```
$ waf --zones=deps

23:53:21 deps deps for src:///comp/waf/demos/qt4/src/window.cpp: ❶
  [src:///comp/waf/demos/qt4/src/window.h, bld:///comp/waf/demos/qt4/src/window.moc]; ❷
  unresolved ['QtGui', 'QGLWidget', 'QWidget'] ❸
```

❶ File being preprocessed

❷ Headers found

❸ System headers discarded

The dependency computation is performed only when the files are not up-to-date, so these commands will display something only when there is a file to compile.

13.3 Library interaction (uselib)

13.3.1 Local libraries

To link a library against another one created in the same Waf project, the attribute *uselib_local* may be used. The include paths, the link path and the library name are automatically exported, and the dependent binary is recompiled when the library changes:

```
def build(bld):
        staticlib = bld(
                features       = 'cc cstaticlib', ❶
                source         = 'test_staticlib.c',
                target         = 'teststaticlib',
                export_incdirs = '.') ❷

        main = bld(
                features       = 'cc cprogram', ❸
                source         = 'main.c',
                target         = 'test_c_program',
                includes       = '.',
                uselib_local   = 'teststaticlib') ❹
```

❶ A static library

❷ Include paths to export for use with uselib_local (include paths are not added automatically). These folders are taken relatively to the current target.

❸ A program using the static library declared previously

❹ A list of references to existing libraries declared in the project (either a python list or a string containing the names space-separated)

The library does not need to actually declare a target. Header-only libraries may be created to add common include paths to several targets:

```
def build(bld):
        bld(
                uselib         = '',
                export_incdirs = '.',
                name           = 'common_includes')

        main = bld(
                features        = 'cc cprogram',
                source          = 'main.c',
                target          = 'some_app',
                uselib_local    = 'common_includes')
```

Note
Include paths are only propagated when *export_incdirs* is provided. The include paths are relative to the path of the task generator that declared them.

Note
The local shared libraries and the uselib variables are propagated transitively, but the static libraries are not.

13.3.2 System libraries

To link an application against various *system libraries*, several compilation flags and link flags must be given at once. To reduce the maintenance, a system called *uselib* can be used to give all the flags at the same time:

```
def configure(conf):
        conf.env.CPPPATH_TEST = ['/usr/include'] ❶

        if sys.platform != win32: ❷
                conf.env.CCDEFINES_TEST = ['TEST']
                conf.env.CCFLAGS_TEST   = ['-O0'] ❸
                conf.env.LIB_TEST       = 'test'
                conf.env.LIBPATH_TEST   = ['/usr/lib'] ❹
                conf.env.LINKFLAGS_TEST = ['-g']
                conf.env.CPPPATH_TEST   = ['/opt/gnome/include']

def build(bld):
        mylib = bld(
                features = 'cc cstaticlib',
                source   = 'test_staticlib.c',
                target   = 'teststaticlib',
                uselib   = 'TEST') ❺

        if mylib.env.CC_NAME == 'gcc':
                mylib.cxxflags = ['-O2'] ❻
```

❶ For portability reasons, it is recommended to use CPPPATH instead of giving flags of the form -I/include. Note that the CPPPATH use used by both c and c++

❷ Variables may be left undefined in platform-specific settings, yet the build scripts will remain identical.

❸ Declare a few variables during the configuration, the variables follow the convention VAR_NAME

❹ Values should be declared as lists excepts for LIB and STATICLIB

❺ Add all the VAR_NAME corresponding to the uselib NAME, which is *TEST* in this example

❻ *Model to avoid*: setting the flags and checking for the configuration must be performed in the configuration section

The variables used for c/c++ are the following:

Table 13.1: Uselib variables and task generator attributes for c/c++

Uselib variable	Attribute	Usage
LIB	lib	list of sharedlibrary names to use, without prefix or extension
STATICLIB	staticlib	list of static library names to use, without prefix or extension
LIBPATH	libpath	list of search path for shared and static libraries
RPATH	rpath	list of paths to hard-code into the binary during linking time
CXXFLAGS	cxxflags	flags to use during the compilation of c++ files
CCFLAGS	ccflags	flags to use during the compilation of c files
CPPPATH	includes	include paths
CXXDEPS		a variable to trigger c++ file recompilations when it changes
CCDEPS		same as above, for c
CCDEFINES,CXXDEFINES	defines	list of defines in the form [*key*=*value*, *key*, . . .]
FRAMEWORK	framework	list of frameworks to use, requires the waf tool *osx*
FRAMEWORKPATH	frameworkpath	list of framework paths to use, requires the waf tool *osx*

The variables may be left empty for later use, and will not cause errors. During the development, the configuration cache files (for example, default.cache.py) may be modified from a text editor to try different configurations without forcing a whole project reconfiguration. The files affected will be rebuilt however.

13.3.3 Specific settings by object files

In some cases, it is necessary to re-use object files generated by another task generator to avoid recompilations. This is similar to copy-pasting code, so it is discouraged in general. Another use for this is to enable some compilation flags for specific files. Although it is not exactly part of the library prossing, the attribute "add_objects" can be used for this purpose:

```
def build(bld):
        some_objects = bld(
                features        = 'cc', ❶
                source          = 'test.c',
                ccflags         = '-O3',
                target          = 'my_objs')

        main = bld(
                features        = 'cc cprogram',
                source          = 'main.c',
                ccflags         = '-O2', ❷
                target          = 'test_c_program',
                add_objects     = 'my_objs') ❸
```

❶ Files will be compiled in c mode, but no program or library will be produced

❷ Different compilation flags may be used

❸ The objects will be added automatically in the link stage

13.4 Configuration helpers

13.4.1 Configuration tests

The method *check* is used to detect parameters using a small build project. The main parameters are the following

1. msg: title of the test to execute
2. okmsg: message to display when the test succeeds
3. errmsg: message to display when the test fails
4. mandatory: when true, raise a configuration exception if the test fails
5. env: environment to use for the build (conf.env is used by default)
6. compile_mode: *cc* or *cxx*
7. define_name: add a define for the configuration header when the test succeeds (in most cases it is calculated automatically)

Besides the main parameters, the attributes from c/c++ task generators may be used. Here is a concrete example:

```
def configure(conf):

        conf.check(header_name=’time.h’, compile_mode=’cc’) ❶
        conf.check_cc(function_name=’printf’, header_name="stdio.h", mandatory=True) ❷
        conf.check_cc(fragment=’int main() {2+2==4;}\n’, define_name="boobah") ❸
        conf.check_cc(lib=’m’, ccflags=’-Wall’, defines=[’var=foo’, ’x=y’],
                uselib_store=’M’) ❹
        conf.check_cxx(lib=’linux’, uselib=’M’, cxxflags=’-O2’) ❺

        conf.check_cc(fragment=’’’
                        #include <stdio.h>
                        int main() { printf("4"); return 0; } ’’’,
                define_name = "booeah",
                execute     = True,
                define_ret  = True,
                msg         = "Checking for something") ❻

        conf.write_config_header(’config.h’) ❼
```

❶ Try to compile a program using the configuration header time.h, if present on the system, if the test is successful, the define HAVE_TIME_H will be added

❷ Try to compile a program with the function printf, adding the header stdio.h (the header_name may be a list of additional headers). The parameter mandatory will make the test raise an exception if it fails. Note that convenience methods *check_cc* and *check_cxx* only define the compilation mode *compile_mode*

❸ Try to compile a piece of code, and if the test is successful, define the name boobah

❹ Modifications made to the task generator environment are not stored. When the test is successful and when the attribute uselib_store is provided, the names lib, cflags and defines will be converted into uselib variables LIB_M, CCFLAGS_M and DEFINE_M and the flag values are added to the configuration environment.

❺ Try to compile a simple c program against a library called *linux*, and reuse the previous parameters for libm (uselib)

❻ Execute a simple program, collect the output, and put it in a define when successful

❼ After all the tests are executed, write a configuration header in the build directory (optional). The configuration header is used to limit the size of the command-line.

Here is an example of a *config.h* produced with the previous test code:

```
/* Configuration header created by Waf - do not edit */
#ifndef _CONFIG_H_WAF
#define _CONFIG_H_WAF

#define HAVE_PRINTF 1
#define HAVE_TIME_H 1
#define boobah 1
#define booeah "4"

#endif /* _CONFIG_H_WAF */
```

The file `default.cache.py` will contain the following variables:

```
CCDEFINES_M = ['var=foo', 'x=y']
CXXDEFINES_M = ['var=foo', 'x=y']
CXXFLAGS_M = ['-Wall']
CCFLAGS_M = ['-Wall']
LIB_M = ['m']
boobah = 1
booeah = '4'
defines = {'booeah': '"4"', 'boobah': 1, 'HAVE_TIME_H': 1, 'HAVE_PRINTF': 1}
dep_files = ['config.h']
waf_config_files = ['/compilation/waf/demos/adv/build/default/config.h']
```

Note
The methods *conf.check* all use task generators internally. This means that the attributes *includes*, *defines*, *cxxflags* may be used (not all shown here).

Note
The attribute *compile_mode* may also contain user-defined task generator features.

13.4.2 Creating configuration headers

Adding lots of command-line define values increases the size of the command-line and conceals the useful information (differences). Some projects use headers which are generated during the configuration, they are not modified during the build and they are not installed or redistributed. This system is useful for huge projects, and has been made popular by autoconf-based projects.

Writing configuration headers can be performed using the following methods:

```
def configure(conf):
        conf.define('NOLIBF', 1)
        conf.undefine('NOLIBF')
        conf.define('LIBF', 1)
        conf.define('LIBF_VERSION', '1.0.2')
        conf.write_config_header('config.h')
```

The code snipped will produce the following *config.h* in the build directory:

```
build/
|-- c4che
|   |-- build.config.py
|   `-- default.cache.py
|-- config.log
`-- default
    `-- config.h
```

The contents of the config.h for this example are

```
/* Configuration header created by Waf - do not edit */
#ifndef _CONFIG_H_WAF
#define _CONFIG_H_WAF

/* #undef NOLIBF */
#define LIBF 1
#define LIBF_VERSION "1.0.2"

#endif /* _CONFIG_H_WAF */
```

13.4.3 Pkg-config

Instead of duplicating the configuration detection in all dependent projects, configuration files may be written when libraries are installed. To ease the interaction with build systems based on Make (cannot query databases or apis), small applications have been created for reading the cache files and to interpret the parameters (with names traditionally ending in *-config*): pkg-config, wx-config, sdl-config, etc.

The method *check_cfg* is provided to ease the interaction with these applications. Here are a few examples:

```
def configure(conf):
        conf.check_cfg(atleast_pkgconfig_version='0.0.0') ❶
        pango_version = conf.check_cfg(modversion='pango', mandatory=True) ❷

        conf.check_cfg(package='pango') ❸
        conf.check_cfg(package='pango', uselib_store='MYPANGO', mandatory=True) ❹

        conf.check_cfg(package='pango',
                args='"pango >= 1.0.0" "pango < 9.9.9" --cflags --libs', ❺
                msg="Checking for pango 1.0.0", mandatory=True) ❻

        conf.check_cfg(path='sdl-config', args='--cflags --libs',
                package='', uselib_store='SDL') ❼
        conf.check_cfg(path='mpicc', args='--showme:compile --showme:link',
                package='', uselib_store='OPEN_MPI')
```

❶ Check for the pkg-config version

❷ Retrieve the module version for a package. The returned object is a string containing the version number or an empty string in case of any errors. If there were no errors, *PANGO_VERSION* is defined, can be overridden with the attribute "uselib_store=*MYPANGO*".

❸ Obtain the flags for a package and assign them to the uselib PANGO (calculated automatically from the package name)

❹ Store the flags to the uselib variable *MYPANGO*, and raise a configuration error if the package cannot be found

❺ Use pkg-config clauses to ensure a particular version number. The arguments and the command are joined into a string and executed by the shell.

❻ Display a custom message on the output.

❼ Obtain the flags for a different configuration program (sdl-config). The example is applicable for other configuration programs such as wx-config, pcre-config, etc

Due to the amount of flags, the lack of standards between config applications, and to the output changing for different operating systems (-I for gcc, /I for msvc), the output of pkg-config is parsed, and the variables for the corresponding uselib are set in a go. The function *parse_flags(line, uselib, env)* in the Waf module config_c.py performs the output analysis.

The outputs are written in the build directory into the file *config.log*:

```
# project cpp_test (0.0.1) configured on Fri Feb 26 20:51:39 2010 by
# waf 1.5.19 (abi 7, python 20602f0 on linux2)
# using /home/waf/bin/waf configure
#
Checking for pkg-config version >= 0.0.0
pkg-config --errors-to-stdout --print-errors --atleast-pkgconfig-version=0.0.0

pkg-config --errors-to-stdout --print-errors --modversion pango

Checking for pango
pkg-config --errors-to-stdout --print-errors  pango

Checking for pango
pkg-config --errors-to-stdout --print-errors  pango

Checking for pango 1.0.0
pkg-config --errors-to-stdout --print-errors "pango >= 1.0.0" --cflags --libs pango

Checking for sdl-config
sdl-config --cflags --libs

Checking for mpicc
mpicc --showme:compile --showme:link
```

After such a configuration, the configuration set contents will be similar to the following:

```
CCFLAGS_OPEN_MPI = ['-pthread']
CPPPATH_OPEN_MPI = ['/usr/lib64/mpi/gcc/openmpi/include/openmpi']
CPPPATH_PANGO = ['/usr/include/pango-1.0', '/usr/include/glib-2.0']
CXXFLAGS_OPEN_MPI = ['-pthread']
HAVE_MYPANGO = 1
HAVE_OPEN_MPI = 1
HAVE_PANGO = 1
LIB_PANGO = ['pango-1.0', 'gobject-2.0', 'gmodule-2.0', 'glib-2.0']
LINKFLAGS_OPEN_MPI = ['-pthread']
PANGO_VERSION = '1.25.3'
PREFIX = '/usr/local'
defines = {'HAVE_OPEN_MPI': 1, 'HAVE_PANGO': 1,
        'PANGO_VERSION': '"1.25.3"', 'HAVE_MYPANGO': 1}
```

Chapter 14

Advanced scenarios

This chapter demonstrates a few examples of the waf library for more complicated and less common scenarios.

14.1 Building the compiler first

The example below demonstrates how to build a compiler which is used for building the remaining targets. The requirements are the following:

1. Create the compiler and all its intermediate tasks
2. Re-use the compiler in a second build step
3. The compiler will transform *.src* files into *.cpp* files, which will be processed too

The first thing to do is to write the expected user script:

```
top = '.'
out = 'build'

def configure(conf):
        conf.check_tool('g++')

def build(bld):
        bld( ❶
                features = 'cxx cprogram',
                source   = 'comp.cpp',
                target   = 'comp',
                name     = 'comp') ❷

        bld.add_group() ❸

        bld(
                features = 'cxx cprogram',
                source   = 'main.cpp a.src', ❹
                target   = 'foo')
```

❶ Build the compiler first, it will result in a binary named *comp*

❷ Give this task generator a name to access it by reference easily

❸ Add a new build group to make certain the compiler is complete before processing the next tasks

4. Provide a file *a.src* to be transformed by *comp* into *a.cpp*

The code to support this scenario may be added in the wscript file or into a waf tool:

```
import Task
Task.simple_task_type('src2cpp',
        '${COMP} ${SRC} ${TGT}', ❶
        color='PINK', before='cxx') ❷

from TaskGen import extension
@extension('.src')
def process_src(self, node): ❸
        tg = self.bld.name_to_obj('comp', self.env) ❹
        tg.post() ❺

        tsk = self.create_task('src2cpp', node, node.change_ext('.cpp')) ❻
        self.allnodes.append(tsk.outputs[0])

        name = tg.link_task.outputs[0].abspath(tg.env) ❼
        tsk.env.COMP = name ❽
```

1. Declare a new task class for processing the source file by our compiler
2. The tasks of this class must be executed before c++ tasks
3. Files of extension *.src* are to be processed by this method
4. Obtain a reference on the task generator producing the compiler
5. Ensure that the compiler tasks are created, else calling `waf --targets=foo` would produce an error
6. Create the task *a.src* → *a.cpp*
7. Get the node corresponding to the program *comp* and obtain its absolute path
8. Bind the path to *comp* for use in the rule

The compilation results will be the following:

```
$ waf distclean configure build -v
'distclean' finished successfully (0.004s)
Checking for program g++,c++              : ok /usr/bin/g++
Checking for program cpp                  : ok /usr/bin/cpp
Checking for program ar                   : ok /usr/bin/ar
Checking for program ranlib               : ok /usr/bin/ranlib
'configure' finished successfully (0.034s)
Waf: Entering directory `/tmp/comp/build'
[1/5] cxx: comp.cpp -> build/default/comp_1.o
16:22:50 runner system command -> ['/usr/bin/g++', '../comp.cpp', '-c', '-o', 'default/ ↩
    comp_1.o']
[2/5] cxx_link: build/default/comp_1.o -> build/default/comp ❶
16:22:50 runner system command -> ['/usr/bin/g++', 'default/comp_1.o', '-o', '/tmp/comp/ ↩
    build/default/comp']
[3/5] src2cpp: a.src -> build/default/a.cpp
16:22:50 runner system command ->  /tmp/comp/build/default/comp ../a.src default/a.cpp ❷
[4/5] cxx: main.cpp -> build/default/main_2.o
16:22:50 runner system command -> ['/usr/bin/g++', '../main.cpp', '-c', '-o', 'default/ ↩
    main_2.o']
[5/5] cxx_link: build/default/main_2.o -> build/default/foo
16:22:50 runner system command -> ['/usr/bin/g++', 'default/main_2.o', '-o', '/tmp/comp/ ↩
    build/default/foo'] ❸
Waf: Leaving directory `/tmp/comp/build'
'build' finished successfully (0.194s)
```

1. Creation of the *comp* program
2. Use the compiler to generate *a.cpp*
3. Compile and link *a.cpp* and *main.cpp* into the program *foo*

Note
In the example above, the task generator *foo* depends on the tasks created by the task generator *tg* (get the path to the binary). Calling *tg.post()* used to force *tg* to create its tasks, and is only called during the processing of *foo*. Doing it from the top-level would break the behaviour of `waf --targets=xyz`

14.2 Mixing extensions and c/c++ features

14.2.1 Files processed by a single task generator

Now let's illustrate the @extension decorator on idl file processing. Files with .idl extension are processed to produce .c and .h files (`foo.idl` → `foo.c` + `foo.h`). The .c files must be compiled after being generated.

First, here is the declaration expected in user scripts:

```
top = '.'
out = 'build'

def configure(conf):
        conf.check_tool('g++')

def build(bld):
        bld(
                features = 'cxx cprogram',
                source   = 'foo.idl main.cpp',
                target   = 'myapp'
                )
```

The file *foo.idl* is listed as a source. It will be processed to *foo.cpp* and compiled and linked with *main.cpp*

Here is the code to support this scenario:

```
import Task
from TaskGen import extension

Task.simple_task_type('idl', 'cp ${SRC} ${TGT}', color='BLUE', before='cxx') ❶

@extension('.idl')
def process_idl(self, node):
        cc_node = node.change_ext('.cpp')
        self.create_task('idl', [node], [cc_node]) ❷
        self.allnodes.append(cc_node) ❸
```

1. The ext_out symbol is to force the idl execution before any c++ file is processed. In practice the rule to use would be like `omniidl -bcxx ${SRC} -C${TGT}`
2. Create the task from the *.idl* extension. If the task produces other files (headers, . . .), more nodes should be added along with *cc_node*.
3. Reinject the c++ node to the list of nodes to process

The compilation results will be the following:

```
$ waf distclean configure build
'distclean' finished successfully (0.002s)
Checking for program g++,c++             : ok /usr/bin/g++
Checking for program cpp                 : ok /usr/bin/cpp
Checking for program ar                  : ok /usr/bin/ar
Checking for program ranlib              : ok /usr/bin/ranlib
'configure' finished successfully (0.033s)
Waf: Entering directory `/tmp/idl/build'
[1/4] idl: foo.idl -> build/default/foo.cc
[2/4] cxx: build/default/foo.cc -> build/default/foo_1.o
[3/4] cxx: main.cpp -> build/default/main_1.o
[4/4] cxx_link: build/default/main_1.o build/default/foo_1.o -> build/default/myapp
Waf: Leaving directory `/tmp/idl/build'
'build' finished successfully (0.119s)
```

Note
The drawback of this declaration is that the source files produced by the idl transformation are used by only one task generator.

14.2.2 Resources shared by several task generators

Now suppose that the idl processing outputs will be shared by several task generators. We will first start by writing the expected user script:

```
top = '.'
out = 'out'

def configure(ctx):
        ctx.check_tool('g++')

def build(bld):
        bld( ❶
                source     = 'notify.idl',
                name       = 'idl_gen')

        bld( ❷
                features   = 'cxx cprogram',
                source     = 'main.cpp',
                target     = 'testprog',
                add_source = 'idl_gen') ❸
```

❶ Process an idl file in a first task generator. Name this task generator *idl_gen*

❷ Somewhere else (maybe in another script), another task generator will use the source generated by the idl processing

❸ Reference the idl processing task generator by the name *idl_gen*. This declaration is is similar to *add_objects*

The code to support this scenario will be the following:

```
import Task
from TaskGen import feature, before, extension
Task.simple_task_type('idl', 'cp ${SRC} ${TGT}', before='cxx') ❶

@extension('.idl')
def compile_idl(self, node): ❷
        c_node = node.change_ext('.cpp')
        self.create_task('idl', node, c_node)
        self.more_source = [c_node] ❸

@feature('*')
@before('apply_core')
def process_add_source(self): ❹
        if not getattr(self, 'add_source', None):
                return

        for x in self.to_list(self.add_source): ❺
                y = self.bld.name_to_obj(x, self.env) ❻
                y.post() ❼

                if getattr(y, 'more_source', None):
                        self.allnodes.extend(y.more_source) ❽
```

1. The idl processing must be performed before any c++ task is executed
2. This is the standard way to process a file by extension
3. Bind the output nodes to a new task generator attribute *more_source*
4. A new task generator method is added to process the attribute *add_source*
5. Process each reference in *add_source*
6. Get the task generator from the reference, and for the current variant. In the example above this is the name *idl_gen*
7. Force the task generator we depend on to create its tasks
8. Add the sources from the other task generator to the current list of source

The task execution output will be very similar to the output from the first example:

```
$ waf distclean configure build
'distclean' finished successfully (0.001s)
Checking for program g++,c++             : ok /usr/bin/g++
Checking for program cpp                 : ok /usr/bin/cpp
Checking for program ar                  : ok /usr/bin/ar
Checking for program ranlib              : ok /usr/bin/ranlib
'configure' finished successfully (0.041s)
Waf: Entering directory `/home/waf/Descargas/blah/out'
[1/4] idl: notify.idl -> out/default/notify.cpp
[2/4] cxx: out/default/notify.cpp -> out/default/notify_2.o
[3/4] cxx: main.cpp -> out/default/main_2.o
[4/4] cxx_link: out/default/notify_2.o out/default/main_2.o -> out/default/testprog
Waf: Leaving directory `/home/waf/Descargas/blah/out'
'build' finished successfully (0.124s)
```

Note
This technique is used by the waf library for processing the *add_objects* attribute.

14.3 Inserting special include flags

A scenario that appears from times to times in C/C++ projects is the need to insert specific flags before others, regardless of how flags are usually processed. We will now consider the following case: execute all c++ compilations with the flag '-I.' in first position (before any other include).

First, a look at the definition of the c++ compilation rule shows that the variable *_CXXINCFLAGS* contains the include flags:

```
cxx_str = '${CXX} ${CXXFLAGS} ${CPPFLAGS} ${_CXXINCFLAGS} ${_CXXDEFFLAGS} ${CXX_SRC_F}${SRC ↩
    } ${CXX_TGT_F}${TGT}'
cls = Task.simple_task_type('cxx', cxx_str, color='GREEN', ext_out='.o', ext_in='.cxx',  ↩
    shell=False)
```

Those include flags are set by the method *apply_obj_vars_cxx*. The trick is then to modify *_CXXINCFLAGS* after that method has been executed:

```
top = '.'
out = 'build'

def configure(conf):
        conf.check_tool('g++')

def build(bld):
        bld.new_task_gen(features='cxx cprogram', source='main.cpp', target='test')

from TaskGen import after, feature

@feature('cxx')
@after('apply_obj_vars_cxx')
def insert_myflags(self):
        self.env.prepend_value('_CXXINCFLAGS', '-I.')
```

A related case is how to add the top-level directory containing a configuration header:

```
@feature('cxx')
@after('apply_obj_vars_cxx')
def insert_myflags(self):
        dir = self.bld.srcnode.relpath_gen(self.path)
        self.env.prepend_value('_CXXINCFLAGS', '-I' + dir)
```

14.4 Simple file transformations

The Waf tool *misc* contains various routines to ease file manipulations such as substituting parameters or executing custom compilers. The objective of this section is to illustrate the principles of the current apis.

The following example illustrates how to produce a pkg-config file from a template:

```
top = '.'
out = 'build'

def configure(conf):
        conf.check_tool('misc')

def build(bld):
        obj = bld(
                features = 'subst',
                source   = 'test.pc.in',
                target   = 'test.pc')
        obj.dict     = {'LIBS': '-lm', 'CFLAGS': '-Wall', 'VERSION': '1.0'}
```

Any kind of map will work for *obj.dict*, for example env.get_merged_dict(). The variables must be declared in the template file by enclosing the names between @ *characters* (m4 syntax):

```
Name: test
Description: @CFLAGS@
Version: @VERSION@
Libs: -L${libdir} @LIBS@
Cflags: -I${includedir} @CFLAGS@
```

The default substitution function may be replaced by changing the attribute *func* of the task generator.

14.5 A compiler producing source files with names unknown in advance

The example below demonstrates how to tackle the following requirements:

1. A compiler *produces source files* (.c files) which are to be processed
2. The source file names are **not known in advance**
3. The task must be **run only if necessary**
4. Other tasks **may depend on the tasks** processing the source produced (compile and link the .c files)

The main difficulty in this scenario is to store the information on the source file produced and to create the corresponding tasks each time.

```
VERSION='0.0.1'
APPNAME='unknown_outputs'
top = '.'
out = 'build'

def configure(conf):
        # used only when configured from the same folder
        conf.check_tool('gcc')
        conf.env.SHPIP_COMPILER = os.getcwd() + os.sep + "bad_compiler.py"

def build(bld):
        staticlib = bld()
        staticlib.features = 'cc cstaticlib'
        staticlib.source = 'x.c foo.shpip' ❶
        staticlib.target='teststaticlib'
        staticlib.includes = '.'


## INTERNAL CODE BELOW ##

import os
import TaskGen, Task, Utils, Build
from TaskGen import taskgen, feature, before, after, extension
from logging import debug
from Constants import *

@taskgen
@after('apply_link')
@extension('.shpip')
def process_shpip(self, node): ❷
        tsk = shpip_task(self.env, generator=self)
        tsk.task_gen = self
        tsk.set_inputs(node)

class shpip_task(Task.Task): ❸
        """
        A special task, which finds its outputs once it has run
        It outputs cpp files that must be compiled too
        """

        color = 'PINK'
        quiet = True ❹
        # important, no link before all shpip are done
        before = ['cc_link', 'cxx_link', 'ar_link_static']

        def __init__(self, *k, **kw):
                Task.Task.__init__(self, *k, **kw)

        def run(self): ❺
                "runs a program that creates cpp files, capture the output to compile them"
                node = self.inputs[0]

                dir = self.generator.bld.srcnode.bldpath(self.env)
                cmd = 'cd %s && %s %s' % (dir, self.env.SHPIP_COMPILER, node.abspath(self.env))
                try:
                        # read the contents of the file and create cpp files from it
                        files = os.popen(cmd).read().strip()
                except:
                        # comment the following line to disable debugging
                        #raise
                        return 1 # error

                # the variable lst should contain a list of paths to the files produced
                lst = Utils.to_list(files)

                # Waf does not know "magically" what files are produced
                # In the most general case it may be necessary to run os.listdir() to see them
                # In this demo the command outputs is giving us this list

                # the files exist in the build dir only so we do not use find_or_declare
                build_nodes = [node.parent.exclusive_build_node(x) for x in lst]
                self.outputs = build_nodes

                # create the cpp tasks, in the thread
                self.more_tasks = self.add_cpp_tasks(build_nodes) ❻

                # cache the file names and the task signature
                node = self.inputs[0]
                sig = self.signature()
                self.generator.bld.raw_deps[self.unique_id()] = [sig] + lst

                return 0 # no error

        def runnable_status(self):
                # look at the cache, if the shpip task was already run
                # and if the source has not changed, create the corresponding cpp tasks

                for t in self.run_after:
                        if not t.hasrun:
                                return ASK_LATER

                tree = self.generator.bld
                node = self.inputs[0]
                try: ❼
                        sig = self.signature()
                        key = self.unique_id()
                        deps = tree.raw_deps[key]
                        prev_sig = tree.task_sigs[key][0]
                except KeyError:
                        pass
                else:
                        # if the file has not changed, create the cpp tasks
                        if prev_sig == sig:
                                lst = [self.task_gen.path.exclusive_build_node(y) for y in  ↩
                                    deps[1:]]
                                self.set_outputs(lst)
                                lst = self.add_cpp_tasks(lst) ❽
                                for tsk in lst:
                                        generator = self.generator.bld.generator
                                        generator.outstanding.append(tsk)


                if not self.outputs:
                        return RUN_ME

                # this is a part of Task.Task:runnable_status: first node does not exist -> ↩
                    run
                # this is necessary after a clean
                env = self.env
                node = self.outputs[0]
                variant = node.variant(env)

                try:
                        time = tree.node_sigs[variant][node.id]
                except KeyError:
                        debug("run task #%d - the first node does not exist" % self.idx, ' ↩
                            task')
                        try: new_sig = self.signature()
                        except KeyError:
                                return RUN_ME

                        ret = self.can_retrieve_cache(new_sig)
                        return ret and SKIP_ME or RUN_ME

                return SKIP_ME

        def add_cpp_tasks(self, lst): ❾
                "creates cpp tasks after the build has started"
                tgen = self.task_gen
                tsklst = []

                for node in lst:
                        TaskGen.task_gen.mapped['c_hook'](tgen, node)
                        task = tgen.compiled_tasks[-1]
                        task.set_run_after(self)

                        # important, no link before compilations are all over
                        try:
                                self.generator.link_task.set_run_after(task)
                        except AttributeError:
                                pass

                        tgen.link_task.inputs.append(task.outputs[0])
                        tsklst.append(task)

                        # if headers are produced something like this can be done
                        # to add the include paths
                        dir = task.inputs[0].parent
                        # include paths for c++ and c
                        self.env.append_unique('_CXXINCFLAGS', '-I%s' % dir.abspath(self. ↩
                            env))
                        self.env.append_unique('_CCINCFLAGS', '-I%s' % dir.abspath(self.env ↩
                            ))
                        self.env.append_value('INC_PATHS', dir) # for the waf preprocessor
                return tsklst
```

1. An example. The source line contains a directive *foo.shpip* which triggers the creation of a shpip task (it does not represent a real file)
2. This method is used to create the shpip task when a file ending in *.shpip* is found
3. Create the new task type
4. Disable the warnings raised because the task has no input and outputs
5. Execute the task
6. Retrieve the information on the source files created
7. Create the c++ tasks used for processing the source files found
8. f the tasks are created during a task execution (in an execution thread), the tasks must be re-injected by adding them to the attribute *more_tasks*
9. If the tasks are created during the task examination (runnable_status), the tasks can be injected directly in the build by using the attribute *outstanding* of the scheduler

Chapter 15

Using the development version

A few notes on the waf development follow.

15.1 Tracing the execution

The generic flags to add more information to the stack traces or to the messages is *-v* (verbosity), it is used to display the command-lines executed during a build:

```
$ waf -v
```

To display all the traces (useful for bug reports), use the following flag:

```
$ waf -vvv
```

Debugging information can be filtered easily with the flag *zones*:

```
$ waf --zones=action
```

Tracing zones must be comma-separated, for example:

```
$ waf --zones=action,envhash,task_gen
```

The Waf module *Logs* replaces the Python module logging. In the source code, traces are provided by using the *debug* function, they must obey the format "zone: message" like in the following:

```
Logs.debug("task: task %r must run as it was never run before or the task code changed" % ↩
    self)
```

The following zones are used in Waf:

Table 15.1: Debugging zones

Zone	**Description**
runner	command-lines executed (enabled when -v is provided without debugging zones)
deps	implicit dependencies found (task scanners)
task_gen	task creation (from task generators) and task generator method execution
action	functions to execute for building the targets
env	environment contents

Table 15.1: (continued)

Zone	Description
envhash	hashes of the environment objects - helps seeing what changes
build	build context operations such as filesystem access
preproc	preprocessor execution
group	groups and task generators

Warning
Debugging information can be displayed only after the command-line has been parsed. For example, no debugging information will be displayed when a waf tool is being by *tool_options*

15.2 Benchmarking

The script `utils/genbench.py` generates a simple benchmark for Waf. The habitual use is the following:

```
$ utils/genbench.py /tmp/build 50 100 15 5
$ cd /tmp/build
$ waf configure
$ waf -p -j2
```

The project created contains 50 libraries with 100 classes for each, each source file having 15 include headers pointing to the same library and 5 headers pointing to the headers of other libraries in the project.

The time taken to create the tasks and to resolve the dependencies can be obtained by injecting code to disable the actual compilation, for example:

```
def build(bld):
        import Task
        def touch_func(task):
                for x in task.outputs:
                        open(x.abspath(task.env), 'w').close()
        for x in Task.TaskBase.classes.keys():
                cls = Task.TaskBase.classes[x]
                cls.func = touch_func
                cls.color = 'CYAN'
```

15.3 Profiling

Profiling information is obtained by calling the module cProfile and by injecting specific code. The two most interesting methods to profile are *flush* and *compile*. The most important number from the profiling information is the amount of function calls, and reducing it results in noticeable speedups. Here is an example on the method compile:

```
import Build
def ncomp(self):
        import cProfile, pstats
        cProfile.run('import Build; Build.bld.rep_compile()', 'profi.txt')
        p = pstats.Stats('profi.txt')
        p.sort_stats('time').print_stats(150)

Build.bld.rep_compile = Build.bld.compile
Build.bld.compile = ncomp
```

Here the output obtained on a benchmark build created as explained in the previous section:

```
Sun Nov  9 19:22:59 2008    prof.txt

         959695 function calls (917845 primitive calls) in 3.195 CPU seconds

   Ordered by: internal time
   List reduced from 187 to 10 due to restriction 10

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.531    0.531    0.531    0.531 {cPickle.dump}
73950/42150    0.223    0.000    0.276    0.000 Environment.py:66(__getitem__)
     5000    0.215    0.000    0.321    0.000 Task.py:770(compute_sig_implicit_deps)
   204434    0.180    0.000    0.180    0.000 {method 'update' of '_hashlib.HASH' objects}
     5050    0.170    0.000    0.551    0.000 Task.py:706(sig_vars)
    10000    0.144    0.000    0.301    0.000 Utils.py:98(h_file)
25255/15205    0.113    0.000    0.152    0.000 Node.py:329(abspath)
    15050    0.107    0.000    0.342    0.000 Task.py:481(unique_id)
    15301    0.102    0.000    0.102    0.000 Environment.py:50(variant)
   132062    0.097    0.000    0.097    0.000 {method 'get' of 'dict' objects}
```

From the profile information, it appears that the hot spots are, in order:

1. The persistence implemented by the cPickle module (the cache file to serialize takes about 3Mb)
2. Accessing configuration data from the Environment instances
3. Computing implicit dependencies (the test project contains lots of interleaved dependencies)

In practice, the time taken by these methods is not significant enough to justify code changes. Also, profiling must be carried out on builds which last at least several seconds.

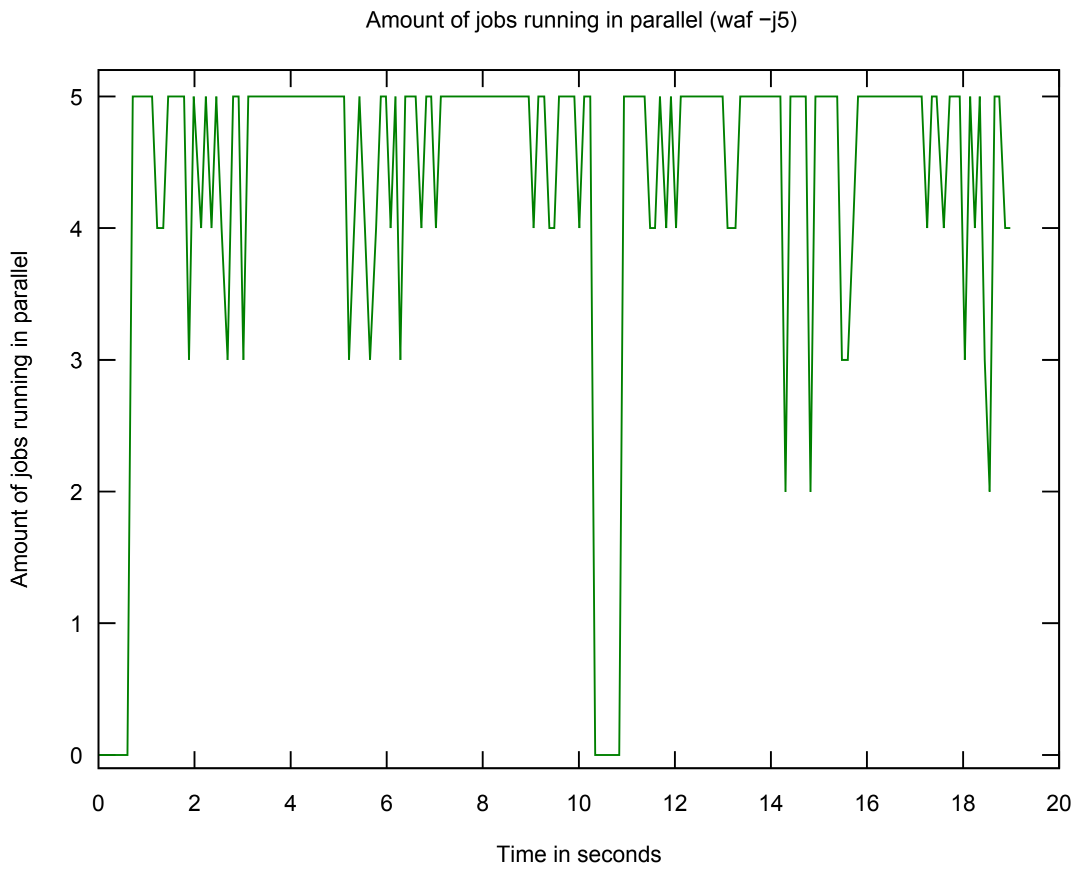
15.4 Tracing parallel task execution

A special Waf tool named *ParallelDebug* is used to inject code in Waf modules and obtain information on the execution. This module is provided in the folder `playground` and must be imported in one's project before use:

```
def configure(conf):
        conf.check_tool('ParallelDebug', tooldir='.')
```

After executing a full build, a trace of the execution is output in `/tmp/test.dat`; it may be processed by other applications such as Gnuplot:

```
set terminal png
set output "output.png"
set ylabel "Amount of jobs running in parallel"
set xlabel "Time in seconds"
set title "Amount of jobs running in parallel (waf -j5)"
unset label
set yrange [-0.1:5.2]
set ytic 1
plot 'test.dat' using 1:3 with lines title "" lt 2
```



15.5 Obtaining the latest source code

Waf is hosted on Google code, and uses Subversion for source control. To obtain the development copy, use:

```
$ svn checkout http://waf.googlecode.com/svn/trunk/ waf-read-only
$ cd waf-read-only
$ ./waf-light --make-waf
```

To avoid regenerating Waf each time, the environment variable WAFDIR can be used to point to the directory containing wafadmin:

```
$ export WAFDIR=/path/to/directory/
```

Although the Waf binary depends on Python 2.3, the Waf source code depends on Python 2.4. When generating Waf, a parser modifies the source code and performs the following operations:

1. Move the decorators into simple function calls
2. Add the imports for the module sets
3. Eliminate redundant spaces
4. Strip comments to reduce the size

15.6 Programming constraints

Though Waf is written in Python, additional restrictions apply to the source code:

1. Identation is tab-only, and the maximum line length should be about 200 characters
2. The development code is kept compatible with Python 2.3, to the exception of decorators in the Tools directory. In particular, the Waf binary can be generated using Python 2.3
3. The `wafadmin` modules must be insulated from the `Tools` modules to keep the Waf core small and language independent
4. Api compatibility is maintained in the cycle of a minor version (from 1.5.0 to 1.5.n)

Note
More code always means more bugs. Whenever possible, unnecessary code must be removed, and the existing code base should be simplified.

Chapter 16

Overview of the Waf architecture

A very open architecture is provided, in which all classes are open for extensions.

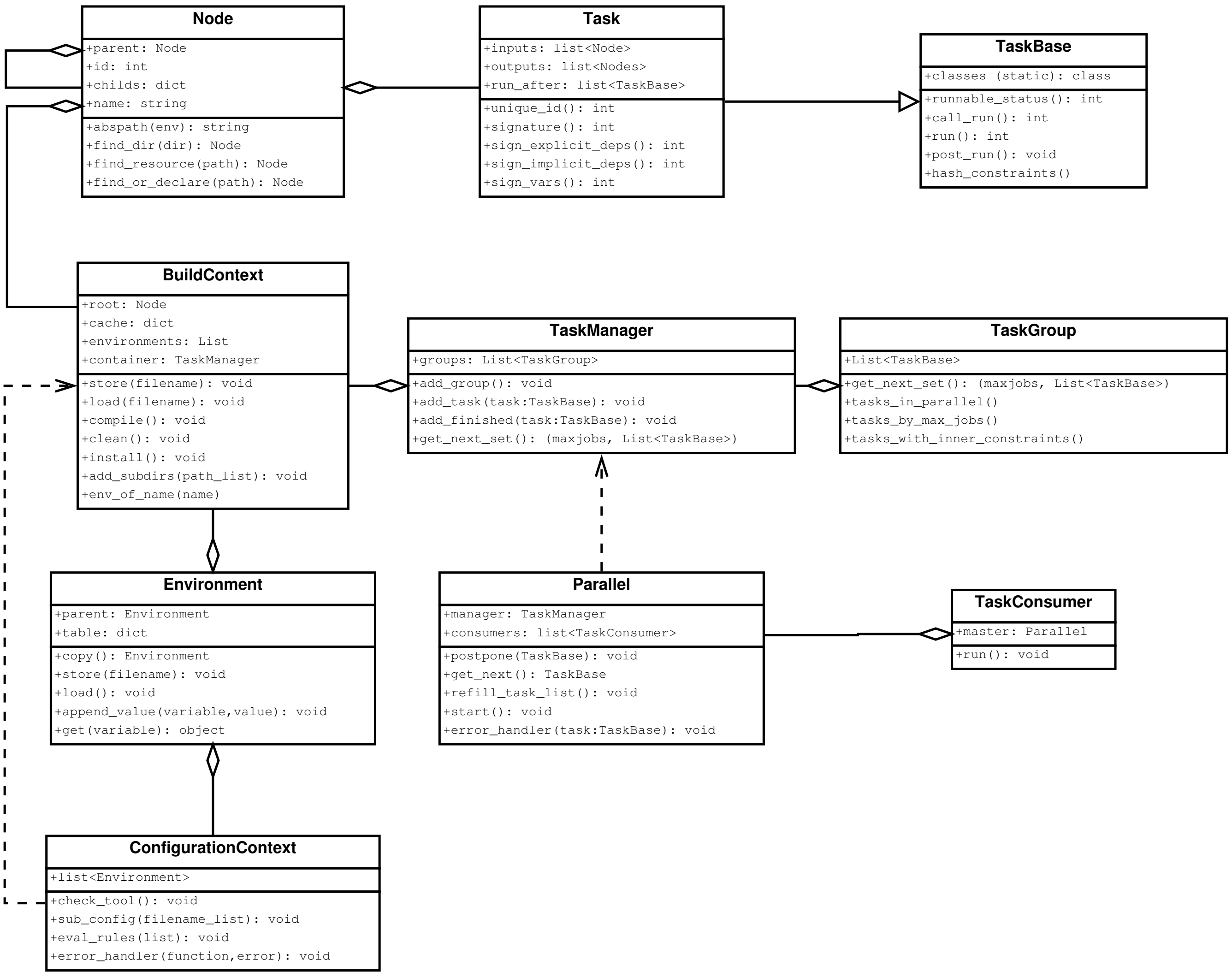
16.1 The core library

Waf is based on only 12 modules which constitute the core library. They are located in the directory `wafadmin/`. The modules located under `wafadmin/Tools` add support for programming languages and more tools, but are not essential for the Waf core.

Table 16.1: The core library

Module	**Role**
Build	Defines the build context class, which holds the data for one build (paths, configuration data)
Configure	Contains the configuration context class, which is used for launching configuration tests, and the extension system
Constants	Provides the constants used in the project
Environment	Contains a dictionary class which supports a lightweight copy scheme and provides persistence services
Logs	Loggging system
Node	Contains the file system representation class
Options	Provides a custom command-line option processing system based on optparse
Runner	Contains the task execution system (thread-based producer-consumer)
Scripting	Constitutes the entry point of the Waf application, executes the user commands such as build, configuration and installation
TaskGen	Provides the task generator system, and its extension system based on method addition
Task	Contains the task classes, and the task containers
Utils	Contains the support functions and classes re-used in other Waf modules

The essential classes and methods from the core library are represented on the following diagram:



16.2 Build context instances

Executing tasks, accessing the file system and consulting the results of a previous build are very different concerns which have to be encapsulated properly. The core class representing a build is a build context.

16.2.1 Build context and persistence

The build context holds all the information necessary for a build. To accelerate the start-up, a part of the information is stored and loaded between the runs. The persistent attributes are the following:

Table 16.2: Build context persistence

Attribute	**Information**
root	Node representing the root of the file system
srcnode	Node representing the source directory
bldnode	Node representing the build directory (rarely used)
node_sigs	File hashes (dict mapping Node ids to hashes)
node_deps	Implicit dependencies (dict mapping Node ids)
raw_deps	Implicit file dependencies which could not be resolved (dict mapping Node ids to lists of strings)

Table 16.2: (continued)

Attribute	Information
task_sigs	Signature of the tasks previously run (dict mapping a Task id to a hash)
id_nodes	Sequence for generating unique node instance ids (id of the last Node created)

16.2.2 Parallelization concerns

Build contexts perform an *os.chdir* call before starting to execute the tasks. When running build contexts within build contexts (tasks), the current working directory may cause various problems. To work around them, it may be necessary to change the compilation rules (compile from the file system root) and to inject code (replace bld.compile).

Direct *Node* instances are not used anywhere in the Waf code. Instead, each build context creates a new Node subclass (bld.node_class), on which the build context instance is attached as a class attribute.

16.2.3 Threading concerns

Nearly all the code is executed in the main thread. The other threads are merely waiting for new tasks, and executing the methods *run* and *install* from the task instances. As a consequence, such methods should contain as little code as possible, and access the BuildContext in a read-only manner. If such tasks must declare new nodes while executing the build (find_dir, find_resource, ..), then locks must be used to prevent concurrent access (concurrent directory and node creation).

If the run methods have to modify the build context, it is recommended to overload the method *get_out* of the scheduler and to execute methods in an event-like manner (data is attached to the task, and the method get_out executes the code). Adding more tasks from a running task is demonstrated <xref linkend="runtime_discovered_outputs"/>.

16.3 Execution overview

16.3.1 File system access

File system access is performed through an abstraction layer formed by the build context and *Node* instances. The data structure was carefully designed to maximize performance, so it is unlikely that it will change again in the future. The idea is to represent one file or one directory by a single Node instance. Dependent data such as file hashes are stored on the build context object and allowed to be persisted. Three kinds of nodes are declared: files, build files and folders. The nodes in a particular directory are unique, but build files used in several variant add duplicate entry on the build context cache.

To access a file, the methods *Node::find_resource*, *Node::find_build* (find an existing resource or declare a build node) and *Node::find_dir* must be used. While searching for a particular node, the folders are automatically searched once for the files. Old nodes (which do not have a corresponding file) are automatically removed, except for the build nodes. In some cases (lots of files added and removed), it may be necessary to perform a *Waf clean* to eliminate the information on build files which do not exist anymore.

16.3.2 Task classes

The whole process of generating tasks through Waf is performed by methods added on the class task_gen by code injection. This often puzzles the programmers used to static languages where new functions or classes cannot be defined at runtime.

The task generators automatically inherit the build context attribute *bld* when created from bld(. . .). Likewise, tasks created from a task generator (create_task) automatically inherit their generator, and their build context. Direct instantiation may result in problems when running Waf as a library.

The tasks created by task generator methods are automatically stored on the build context task manager, which stores the task into a task group. The task groups are later used by the scheduler to obtain the task which may run (state machine). Target (un)installation is performed right after a task has run, using the method *install*.

16.4 Performance and build accuracy

From the experience with tools such as SCons, users may be concerned about performance and think that all build systems based on interpreted languages such as Python would not scale. We will now describe why this is not the case for Waf and why Waf should be chosen for building very large projects.

16.4.1 Comparing Waf against other build systems

Since Waf considers the file contents in the build process, it is often thought that Waf would be much slower than make. For a test project having 5000 files (generated from the script located in `tools/genbench.py`), on a 1.5Ghz computer, the Waf runtime is actually slightly faster than the Gnu/Make one (less than one second). The reason is the time to launch a new process - make is usually called recursively, once by directory.

For huge projects, calling make recursively is necessary for flexibility, but it hurts performance (launch many processes), and CPU utilization (running tasks in parallel). Make-based build systems such as CMake or Autotools inherit the limitations of Make.

Though Waf uses a similar design as SCons, Waf is about 15 times faster for similar features and without sacrificing build accuracy. The main reasons for this are the following:

1. The Waf data structures (file system representation, tasks) have been carefully chosen to minimize memory usage and data duplication
2. For a project of the same size, SCons requires at least 10 times as many function calls

A few benchmarks are maintained at this location

16.4.2 Waf hashing schemes and build accuracy

To rebuild targets when source file change, the file contents are hashed and compared. The hashes are used to identify the tasks, and to retrieve the files from a cache (folder defined by the environment variable *WAFCACHE*). Besides command-lines, this scheme also takes file dependencies into account: it is more accurate than caching systems such as *ccache*.

The Waf hashing scheme uses the md5 algorithm provided by the Python distribution. It is fast enough for up to about 100Mb of data and about 10000 files and very safe (virtually no risk of collision).

If more than 100Mb of data is present in the project, it may be necessary to use a faster hashing algorithm. An implementation of the fnv algorithm is present in the Waf distribution, and can replace md5 without really degrading accuracy.

If more than 10000 files are present, it may be necessary to replace the hashing system by a *file name+size+timestamp hash scheme*. An example is provided in the comment section of the module `Utils.py`. That scheme is more efficient but less accurate: the Waf cache should not be used with this scheme.

Chapter 17

Further reading

Due to the amount of features provided by Waf, this book is forcibly incomplete. For greater understanding and practice the following links are recommended to the reader:

Table 17.1: Recommended links

Link	Description
http://freehackers.org/~tnagy/wafdoc/index.html	The apidocs
http://code.google.com/p/waf	The Waf project page
http://code.google.com/p/waf/w/list	The Waf wiki, including the frequently asked questions (FAQ)
http://groups.google.com/group/waf-users	The Waf mailing-list

Chapter 18

Glossary

Task generator

A task generator is an object instance of the class Task.task_gen. The task generators encapsulate the creation of various task instances at a time, and simplify the creation of ordering constraints between them (for example, compilation tasks are executed before link tasks).

Task

A Waf task is an object instance of the class Task.TaskBase. Waf tasks may be simple (Task.TaskBase) or related to the filesystem (Task.Task). Tasks represent the production of something during the build (files in general), and may be executed in sequence (with ordering constraints) or in parallel.

Tool

A Waf tool is a python module containing Waf-specific extensions. The Waf tools are located in the folder `wafadmin/Tools/` and usually contain a global variable *detect* which may reference functions to execute in the configuration.

Node

The Node class is a data structure used to represent the filesystem in an efficient manner. The node objects may represent source files, folders, or build files. Non-directory nodes may associated to signatures containing the source file hash for source nodes or the task signature that produced the node.