The cool TEX automation tool

User Manual

Version 1.0.1

Prologue

Knowledge brings fear.

From a Futurama episode

Dear reader, please be warned. At first, arara was written for helping me with my LaTeX projects. To be honest, I never intended to release it to the whole world, since I wasn't sure if other people could benefit from arara's features. After all, there's already a plethora of tools available to the TeX community in general. The reason I decided to make arara publicly available is quite simple: I want to contribute to the TeX community, and I want to give my best to make it even more awesome.

That said, here comes the friendly warning: Hic Sunt Dracones. arara is far from being bug-free. I don't even dare telling that the code is stable – although I actually think it is. Besides, you will see that arara gives you enough rope. In other words, *you* will be responsible for how arara behaves and all the consequences from your actions. Sorry to sound scary, but I really needed to tell you this. After all, one of arara's features is the freedom it offers. But as you know, freedom always comes at a cost. Please, don't send me angry letters – or emails, perhaps.

Feedback is surely welcome for me to improve this humble tool, just write an e-mail to `cereda@users.sf.net` and I'll reply as soon as possible. The source code is fully available at `http://github.com/cereda/arara`, feel free to contribute to the project by forking it or sending pull requests. If you want to support LaTeX development by a donation, the best way to do this is donating to the TeX Users Group. Please also consider joining our TeX community at StackExchange.

Paulo Roberto Massa Cereda
The author

Special thanks

I'd like to thank some friends that made arara possible:

Andrew Stacey
: for testing arara, providing great user cases, and for suggesting improvements to the program.

Enrico Gregorio
: for reviewing the original manual and providing great ideas and suggestions to the manual and to the program itself.

Joseph Wright
: for testing it, providing contributed code for Linux and Mac installations, and also blogging about arara in his personal blog.

Marco Daniel
: for heavily testing arara, suggesting enhancements to the manual and to the program itself and also providing lots of contributed rules for common tasks. Marco is now an official collaborator and is helping me a lot with the project management. I have no words to express my gratitude for what Marco has been doing to arara.

Patrick Gundlach
: for advertising arara in the official Twitter channel of Dante – the German TEX User Group.

And at last but not least, I want to thank you, dear reader and potential user, for giving arara a try. It's been an honour to serve the TEX community.

Release information

1.0.1

new Added support for `.tex`, `.dtx` and `.ltx` files. When no extension is provided, arara will automatically look for these extensions in this specific order.

new Added the `--verbose` flag to allow printing the complete log in the terminal. A short `-v` tag is also available. Both `stdout` and `stderr` are printed.

fixed Fixed exit status when an exception is thrown. Now arara also returns a non-zero exit status when something wrong happened. Note that this behaviour happens only when arara is processing a file.

1.0

new First public release.

License

arara is licensed under the New BSD License. It's important to observe that the New BSD License has been verified as a GPL-compatible free software license by the Free Software Foundation, and has been vetted as an open source license by the Open Source Initiative.

Contents

I For users **1**

1 Introduction **3**
1.1 What is arara? . . . 3
1.2 Features . . . 4
1.3 Common uses . . . 5

2 Installation **7**
2.1 Obtaining arara . . . 7
2.2 Deployment . . . 7
2.3 Updating . . . 9

3 Getting started **11**
3.1 Rules . . . 11
3.2 Directives . . . 13
3.3 Orb tags . . . 14
3.4 Examples . . . 16

4 When something goes wrong **25**
4.1 arara messages . . . 25
4.2 Logging . . . 26

5 Best practices **29**
5.1 Plain rules . . . 29
5.2 Directives . . . 30

6 Predefined rules **31**
`latex` . . . 32
`pdflatex` . . . 33
`lualatex` . . . 34
`xelatex` . . . 35
`biber` . . . 35

`bibtex` 36
`makeglossaries` 36
`makeindex` 37
`nomencl` 37
`make` 38
`dvips` 38
`ps2pdf` 39
`removehelpfiles` 39
`cleanhelpfiles` 40

II For developers **41**

7 Writing compiled rules **43**

List of Codes

1 `mydoc.tex` . . . 4
2 `myarticle.tex` . . . 5
3 `mymanual.tex` . . . 6
4 Checking if `java` is installed. . . . 8
5 `arara.cmd` for Windows. . . . 8
6 `arara` for Linux and Mac. . . . 9
7 Testing if arara is correctly deployed. . . . 9
8 `makefoo.yaml`, a basic structure of a plain arara rule. . . . 12
9 `makebar.yaml`, a rule with arguments. . . . 13
10 Example of directives in a `.tex` file. . . . 14
11 `pdflatex.yaml`, first attempt. . . . 16
12 `helloworld.tex` . . . 17
13 arara output for `pdflatex`. . . . 17
14 `pdflatex.yaml`, second attempt. . . . 18
15 `pdflatex.yaml`, third attempt. . . . 19
16 `makeindex.yaml`, first attempt. . . . 19
17 `makeindex.yaml`, second attempt. . . . 20
18 `helloindex.tex` . . . 20
19 Running `helloindex.tex`. . . . 21
20 List of auxiliary files after running `arara helloindex`. . . . 22
21 `clean.yaml` . . . 22
22 `helloindex.tex` with the new `clean` directive. . . . 23
23 Running `helloindex.tex` with the new `clean` rule. . . . 23
24 `arara.log` from `arara helloindex --log`. . . . 27

Part I

For users

Chapter 1

Introduction

Pardon me while I fly my aeroplane.

From a Monty Python sketch

Welcome to arara! I'm glad you were not intimidated by the threatening message in the prologue – What prologue? Anyway, this chapter is a quick introduction to what you can expect from arara. Don't be afraid, it will be easy to digest, I promise.

1.1 What is arara?

Good question. I've been asking it myself for a quite long time. Since I have to provide an official definition for arara– I'm the author, the one to blame – I'd go with something along these lines:

> arara is a TEX automation tool. But maybe not in the traditional sense, such as existing tools like latexmk and rubber. Think of arara as a personal assistant. It is as powerful as you want it to be. arara doesn't provide solutions out of the box, but it gives you subsidies to enhance your TEX experience.

Well, that was a shot in the dark. I'm sorry for this crude definition, but the truth is: arara is generic enough to rely on different schemes. arara will execute what you tell it to execute. How will arara do this? That's the problem: you are in control, so it depends on you.

Maybe I should provide an example for a better understanding. Consider the LATEX code presented in Code 1. How would you compile `mydoc.tex` in

Code 1 • mydoc.tex

```
1 \documentclass{article}
2
3 \begin{document}
4
5 Hello world.
6
7 \end{document}
```

rubber, for instance? It's quite easy, a simple rubber --pdf mydoc would do the trick. Now, if you try arara mydoc, I'm afraid nothing will be generated. Why? Isn't arara supposed to be a TEX automation tool? Well, arara doesn't know what to do with your file. You need to tell it. For now, please understand that you need to provide the batteries for arara to run – bad analogy perhaps, but that's true. Don't worry, we will come back to this example later in the manual and see how to make arara produce the desired output.

As I keep saying since the first pages of this manual: you are in control of your documents. arara won't do anything unless you teach it how to do a task and explicitly tell it to execute the task. Introducing the arara terminology:

How can I teach arara to do a task?
You need to define arara rules.

How can I tell arara to execute a task?
You need to use arara directives.

That's probably one of the major differences of arara from other automation tools. With latexmk and rubber, for example, you have great features out of the box, ready for you to use and abuse – batteries included. arara takes a minimalist approach and gives you the simplicity of doing exactly what you want it to do. Nothing more, nothing less.

1.2 Features

There's nothing so special with arara. It does exactly what you tell it to do. On the other hand, one of the features I like in arara is the ability to

write rules in a human-readable format. You don't need to rely on other formats. Actually, you can write a compiled rule, but I'm almost sure you will never need one.

I like to be in control of my TEX documents – including running commands many times my heart desires and in the order I want. I can create a complex workflow and arara will handle it for me – again, as long as I have the proper rules.

Another feature worth mentioning is the fact that arara is platform independent. I wrote it in Java, so arara runs on top of a Java Virtual Machine (JVM), available on all the major operating systems – in some cases, you might need to install the proper JVM. I tried to keep my code and libraries compatible with older virtual machines – currently, Java 5, 6, 7 and OpenJDK 6 are supported. But beware, if you write system-specific rules, you will need to adapt them when porting to other operating system.

1.3 Common uses

arara can be used in complex workflows, like theses and books. You can tell arara to compile the document, generate indices and apply styles, remove temporary files, compile other `.tex` documents, run MetaPost or MetaFont, create glossaries, call `pdfcrop`, move files, and much more. It's up to you.

Code 2 contains the arara workflow I used for an article I recently wrote. Note that the first call to `pdflatex` creates the `.aux` file, then `bibtex` will extract the cited publications. The next calls to `pdflatex` will insert and refine the references.

Code 2 • `myarticle.tex`

```
% arara: pdflatex
% arara: bibtex
% arara: pdflatex
% arara: pdflatex
\documentclass[journal]{IEEEtran}
...
```

Code 3 contains another arara workflow I used for a manual. I had to use a package that required shell escape, so the calls to `pdflatex` had to enable it. Also, I had an index with a custom formatting, then `makeindex` was called with the proper style.

Code 3 • mymanual.tex

```
% arara: pdflatex: { shell: yes }
% arara: makeindex: { style: mystyle }
% arara: pdflatex: { shell: yes }
% arara: pdflatex: { shell: yes }
\documentclass{book}
...
```

Other workflows can be easily created. There can be an arbitrary number of instructions for arara to execute, so feel free to come up with your own workflow. arara will handle it for you.

I really hope you like my humble contribution to the TEX community. Let arara enhance your TEX experience. Have a good read.

Welcome to arara!

Trivia

Arara is a bird, also known as macaw, native to Mexico, Central America, South America, and formerly the Caribbean. Why did I chose this name? Well, araras are colorful, noisy, naughty and very funny. Everybody loves araras. So why can't you love a tool with the very same name?

Chapter 2

Installation

I would like to buy a hamburger.

Inspector Jacques Clouseau,
The Pink Panther (2006)

Splendid, so you decided to give arara a try? This chapter will cover the installation procedure. Well, to be honest, there is nothing much to see here. The provided `.jar` file is a self-contained, batteries-included executable Java file, so it's not an installation per se, but more like a deployment.

2.1 Obtaining arara

First of all, we need to obtain the arara binary. Go to the project repository and download the current release. The filename to get is `arara.jar`. You can also build arara from sources, but it's far beyond the scope of this manual.

2.2 Deployment

The first step is to create an application folder. Feel free to create a folder anywhere in your computer. It can be `C:\arara`, `/opt/arara` or another location. My setup is usually `C:\paulo\softwares\arara` for Windows machines and `/opt/paulo/arara` for Linux and Mac, but this is of course a matter of personal taste. For convenience, this folder will be called `ARARA_HOME`. Although it's not mandatory, try to avoid folders structures with spaces in the path.

Since arara is written in Java, it requires a Java Virtual Machine. Do you have one installed? If you are not sure, try running `java -version` in the terminal and see if you get an output similar to the Code 4.

Code 4 • Checking if `java` is installed.

```
$ java -version
java version "1.6.0_24"
OpenJDK Runtime Environment (IcedTea6 1.11.1)
OpenJDK Client VM (build 20.0-b12, mixed mode)
```

If you don't have a proper Java Virtual Machine installed, I suggest you to visit the Java website and download one, according to your operating system. Installation instructions are also provided.

Now, copy the `.jar` file we have downloaded in the very first step inside the `ARARA_HOME` folder. Don't forget to add `ARARA_HOME` to the system path. Unfortunately, this manual can't cover the path settings, since it's again a matter of personal taste. For my tools, I usually set the path in my local `.bashrc` with `export PATH="$PATH:$ARARA_HOME"`. It's up to you.

The last step for deploying arara is platform-specific. In order to run arara, we should run `java -jar arara.jar`, but that is not intuitive. To make our lives easier, we will create a shortcut for this command. If you are in Windows, create a file named `arara.cmd` inside `ARARA_HOME` and add the content from Code 5. If you are in Linux or Mac, create a file named `arara` inside `ARARA_HOME` and add the content from Code 6. In Linux and Mac, there only one more thing to do: don't forget to add execute permissions for `arara` by running `chmod +x arara`.

Did you add `ARARA_HOME` to the path? If so, we are good to go. Try running `arara` in the terminal and see if you get the output shown in Code 7.

If the terminal doesn't display the arara logo and usage, please review the deployment steps. Every step is important in order to make arara available in your system. If you have any doubts, feel free to contact me.

Code 5 • `arara.cmd` for Windows.

```
@echo off
java -jar "%~dp0\arara.jar" %*
```

Code 6 • `arara` for Linux and Mac.

```
#!/bin/bash
java -jar "$(dirname "$0")/arara.jar" $*
```

Code 7 • Testing if arara is correctly deployed.

```
$ arara
  __ _ _ __ __ _ _ __ __ _
 / _` | '__/ _` | '__/ _` |
| (_| | | | (_| | | | (_| |
 \__,_|_| \__,_|_| \__,_|

Arara 1.0.1 - The cool TeX automation tool
Copyright (c) 2012, Paulo Roberto Massa Cereda
All rights reserved.

usage: arara [file [--log] [--verbose] | --help |
    --version]

 -h,--help  print the help message
 -l,--log   generate a log output
 -v,--verbose print the command output
 -V,--version print the application version
```

2.3 Updating

If there is a newer version of arara, simply download the `.jar` file and move it inside the `ARARA_HOME` folder, replacing the old one. No further steps are needed, the newer version is already deployed. Try running `arara` in the terminal and see if the version is equal to the one you have downloaded.

Chapter 3

Getting started

Is Batman a scientist?

Homer Simpson

Time for our first contact with arara! It's important to understand two concepts in which arara is based: rules and directives. A *rule* is a formal description of how arara should handle a certain task. For example, if we want to use `pdflatex` with arara, we should have a rule for that. Once a rule is defined, arara automatically provides an access layer to that rule through directives. A *directive* is a special comment in the `.tex` file which will tell arara how it should execute a certain task. A directive can have as many parameters as its corresponding rule has. Don't worry, let's get started with these new concepts.

3.1 Rules

Do you remember `mydoc.tex` from Code 1? When we tried to mimic `rubber` and run `arara mydoc`, nothing happened. We should tell arara how it should handle this execution. Let's start with the rules.

A rule is a plain text file written in the YAML format. I chose this format because it's cleaner and more intuitive to use than other markup languages, besides of course being a data oriented format. As a bonus, YAML rhymes with the word *camel*, so arara is environmentally friendly.

The rules must be placed in a special folder inside `ARARA_HOME`. The full path for plain arara rules is `ARARA_HOME/rules/plain`, so feel free to create this folder structure before proceeding with the reading. Wait a minute, what is a plain rule? Easy, it's a rule written using the YAML format. We

can also have compiled rules in the form of `.jar` files to be placed inside `ARARA_HOME/rules/compiled`, but I'm almost sure you will never need to write one of them. This manual doesn't cover compiled rules, please refer to the developer manual for further reference.

The basic structure of a plain arara rule is presented in Code 8.

Code 8 • `makefoo.yaml`, a basic structure of a plain arara rule.

```
1 !config
2 identifier: makefoo
3 name: MakeFoo
4 command: makefoo @{file}
5 arguments: []
```

The `!config` keyword (line 1) is mandatory and it must be the first line of a plain arara rule. The following keys are defined:

identifier

This key (line 2) acts as a unique identifier for the rule. It's highly recommended to use lowercase letters without spaces, accents or punctuation symbols. As a convention, if you have an identifier named `makefoo`, the rule filename must be `makefoo.yaml`.

name

The `name` key (line 3) holds the name of the task. When running arara, this value will be displayed in the output. In our example, arara will display `Running MakeFoo` when dealing with this task.

command

This key (line 4) contains the system command to be executed. It's highly recommended to avoid interactive commands. Prefer those commands that run as a non-interactive mode, mainly because arara won't output anything in the terminal neither provide user interaction. You probably noticed a strange element `@{file}`: this element is called *orb tag*. For now, just admit they exist. We will come back to them later on, in Section 3.3.

arguments

The `arguments` key (line 5) is a list. In our example, it has an empty list, denoted as `[]`. You can define as many arguments as your command requires. Check Code 9 for an example of a list of arguments.

For more complex rules, we might want to use arguments. Code 9 presents a new rule which makes use of them instead of an empty list as we saw in Code 8.

Code 9 • `makebar.yaml`, a rule with arguments.

```
1 !config
2 identifier: makebar
3 name: MakeBar
4 command: makebar @{one} @{two} @{file}
5 arguments:
6 - identifier: one
7   flag: -i @{value}
8 - identifier: two
9   flag: -j @{value}
```

For every argument in the list, we have a - mark and the proper indentation. The required keys for an argument are:

identifier
: This key (lines 6 and 8) acts as a unique identifier for the argument. It's highly recommended to use lowercase letters without spaces, accents or punctuation symbols.

flag
: The `flag` key (lines 7 and 9) represents the argument value. Please note that we have another orb tag in the definition, `@{value}`. We will discuss them later in Section 3.3.

For now, just keep in mind that arara uses rules to tell it how to do a certain task. In the next sections, when more concepts are presented, we will come back to this subject. Just a taste of things to come: directives are mapped to rules through orb tags. Don't worry, I'll explain how things work.

3.2 Directives

A directive is a special comment inserted in the `.tex` file in which you indicate how arara should behave. You can insert as many directives as you want, and in any position of the `.tex` file. arara will read the whole

Code 10 • Example of directives in a `.tex` file.

```
1 % arara: makefoo
2 % arara: makebar: { one: hello, two: bye }
3 % arara: makebar
4 \documentclass{article}
5 ...
```

file and extract the directives. A directive should be placed in a line of its own, in the form `% arara: <directive>`. There are two types of directives:

empty directive
: An empty directive has only the rule identifier, as we seen in Section 3.1. Lines 1 and 3 of Code 10 show an example of empty directives. Note that you can supress arguments (line 3 in constrast to line 2), but we will see that arara assumes that you know exactly what you are doing.

parametrized directive
: A parametrized directive has the rule identifier followed by its arguments. Line 2 of Code 10 shows an example of a parametrized directive. Note that the arguments are mapped by their identifiers, not by their positions.

The arguments are defined according to the rule mapped by the directive. For example, the rule `makebar` (Code 9) has a list of two arguments, `one` and `two`. So you can safely write `makebar: { one: hello }`, but trying to map a nonexisting argument with `makebar: { three: hi }` will raise an error.

If you want to disable an arara directive, there's no need of removing it from the `.tex` file. Simply replace `% arara:` by `% !arara:` and this directive will be ignored.

Directives are mapped to rules. In Section 3.3 we will learn about orb tags and then revisit rules and directives. I hope the concepts will be clearer since we understand what an orb tag is and how it works.

3.3 Orb tags

When I was planning the mapping scheme, I opted for a templating mechanism. I was looking for flexibility, so the MVEL library was perfect for

the job. I could extend my mapping plans by using the orb tags. An orb tag consists of a `@` character followed by braces `{...}` which contain regular MVEL expressions. In particular, arara uses the `@{}` expression orb, which contains a value expression which will be evaluated to a string, and appended to the output template. For example, the following template `Hello, my name is @{name}` with `name` resolving to `Paulo` will be expanded to `Hello, my name is Paulo`.

When mapping rules, every command argument will be mapped to the form `@{identifier}` with value equals to the content of the `flag` key. There are two reserved orb tags, `@{file}` and `@{value}`. The first one refers to the `.tex` filename argument passed to arara. The extension is removed, so no matter if arara is called with `arara mydoc.tex` or `arara mydoc`, `@{file}` will be expanded to `mydoc`. The `@{file}` value can be overriden, but we will discuss it later. The second reserved orb tag `@{value}` is expanded to the argument value passed in the directive. If you have `makebar: { one: hello }`, the `flag` key of argument `one` will be expanded from the original definition `-i @{value}` to `-i hello`. Now `@{one}` contains the expanded `flag` value, which is `-i hello`. All arguments tags are expanded in the rule command. If one of them is not defined in the directive, arara will admit an empty value, so the `command` flag will be expanded to `makebar -i hello mydoc`. The whole procedure is summarized as follows:

1. arara processed a file named `mydoc.tex`.
2. A directive `makebar: { one: hello }` was found, so arara will look up the rule `makebar.yaml` (Code 9).
3. The argument `one` is defined and has value `hello`, so the corresponding `flag` key will have the orb tag `@{value}` expanded to `hello`. The new value is now added to the template referenced by the `command` key and then `@{one}` is expanded to `-i hello`.
4. The argument `two` is not defined, so the template referenced by the `command` key has `@{two}` expanded to an empty string.
5. There are no more arguments, so the template referenced by the `command` key now expands `@{file}` to `mydoc`.
6. The final command is now `makebar -i hello mydoc`.

There's a reserved directive key named `files`, which is in fact a list. In case you want to override the default `@{file}` value, use the `files` key,

like `makebar: { files: [ thedoc ] }`. This will result in `makebar thedoc` instead of `makebar mydoc`.

If you provide more than one file in the list, arara will replicate the directive for every file found, so `makebar: { files: [ a, b, c ] }` will result in three commands: `makebar a`, `makebar b` and `makebar c`.

3.4 Examples

Now that we know about rules, directives and orb tags, it's time to come up with some examples. I know it's not trivial to understand how arara works, but I'm sure the examples will help with the concepts. Please note that there might have platform-specific rules, so double-check the commands before running them.

PDFLaTeX

Our first example is to add support to `pdflatex`. My first attempt to write this rule is presented in Code 11.

Code 11 • `pdflatex.yaml`, first attempt.

```
1 !config
2 identifier: pdflatex
3 name: PDFLaTeX
4 command: pdflatex -interaction=nonstopmode @{file}.tex
5 arguments: []
```

So far, so good. The `command` flag has the `pdflatex` program and also the flag `-interaction=nonstopmode`, since arara doesn't provide user interaction. Now we can add the `pdflatex` directive to our `.tex` file, as we can see in Code 12.

It's just a matter of calling `arara helloworld.tex` (you can also provide the `.tex` extension by calling `arara helloworld.tex`) and arara will process our file, according to the Code 13.

Great, our first rule works like a charm. Once we define a rule, the directive is automatically available for us to call it as many times as we want. What if we make this rule better? Consider the following situation:

> Sometimes, we need to use `\write18` or call a package that makes use of it (for example, `minted`). It's very dangerous to en-

Code 12 • helloworld.tex

```
% arara: pdflatex
\documentclass{article}

\begin{document}

Hello world.

\end{document}
```

Code 13 • arara output for pdflatex.

```
$ arara helloworld
  __ _ _ __ __ _ _ __ __ _
 / _` | '__/ _` | '__/ _` |
| (_| | | | (_| | | | (_| |
 \__,_|_| \__,_|_| \__,_|

Running PDFLaTeX... SUCCESS
```

able shell escape globally, but changing the pdflatex call every time we need it sounds boring.

arara has a special treatment for cases like this. In the early stages of development, arara was able to handle boolean values. Entries with true or false, on or off, yes and no were mapped to boolean values. If you wanted to use yes as text, you could explicitly tell arara that the value was a string by enclosing it with single or double quotes, 'yes' or "yes". In my humble opinion, it was a good design at first, but it opened a dangerous pitfall: if a certain mapping was expecting a boolean, but another value was received, the result was automatically resolved to true. We have enough problems of arara itself giving us enough rope, so I decided to consider every argument value as string. No big deal, we can still mimic a boolean behaviour, as we will see in our next attempt.

We will rewrite our pdflatex rule to include a flag for shell escape. Another cool feature will be presented now, as we can see in the new rule shown in Code 14.

Code 14 • `pdflatex.yaml`, second attempt.

```
1 !config
2 identifier: pdflatex
3 name: PDFLaTeX
4 command: pdflatex -interaction=nonstopmode @{shell} @{file}.
     tex
5 arguments:
6 - identifier: shell
7   flag: '@{value == "yes" ? "-shell-escape" : "-no-shell-
       escape" }'
```

Orb tags allow evaluation inside the tag block! Line 7 from Code 14 makes use of the ternary operator `?:` which defines a conditional expression. In the first part of the evaluation, we check if `value` is equal to the string `"yes"`. If so, `"-shell-escape"` is defined as the result of the operation. If the conditional expression is false, `"-no-shell-escape"` is set instead.

What if you want to allow `true` and `on` as valid options as well? We can easily rewrite our orb tag to check for additional values. It's also possible to invoke some string methods on orb tags, like `toLowerCase`. A third attempt is presented in Code 15. The `toLowerCase` method was added to allow entries like `Yes`, `yEs` and other combinations. Although arara can support cases in arguments and values, I recommend you to stick with lowercase entries. By the way, for more complex orb tag schemes, it's important to enclose the orb tags with either single or double quotes. Of course, if you use single quotes to enclose the orb tags, use double quotes for internal evaluations, and vice versa.

With this new rule, it's now easy to enable the shell escape option in `pdflatex`. Simply go with the directive `pdflatex: { shell: yes }`. You can also use `true` or `on` instead of `yes`. Any other value for `shell` will disable the shell escape option. It's important to observe that arara directives have no mandatory arguments. If you want to add a dangerous option like `-shell-escape`, consider calling it as an argument with a proper check and rely on a safe state for the argument fallback.

MakeIndex

For the next example, we will create a rule for `makeindex`. To be honest, although `makeindex` has a lot of possible arguments, I only use the `-s` flag

Code 15 • `pdflatex.yaml`, third attempt.

```
!config
identifier: pdflatex
name: PDFLaTeX
command: pdflatex -interaction=nonstopmode @{shell} @{file}.
    tex
arguments:
- identifier: shell
  flag: '@{value.toLowerCase() == "yes" || value.toLowerCase
      () == "true" || value.toLowerCase() == "on" ? "-shell-
      escape" : "-no-shell-escape" }'
```

once in a while. Code 16 shows our first attempt of writing this rule.

Code 16 • `makeindex.yaml`, first attempt.

```
!config
identifier: makeindex
name: MakeIndex
command: makeindex @{style} @{file}.idx
arguments:
- identifier: style
  flag: -s @{value}
```

As a follow-up to our fist attempt, we will now add support for the `-g` flag that employs German word ordering in the index. Since this flag is basically a switch, we can borrow the same tactic used for shell escape in the previous example. The new rule is presented in Code 17.

The new `makeindex` rule presented in Code 17 looks good. We can now test the compilation workflow with an example. Consider a file named `helloindex.tex` which has a few index entries, presented in Code 18. As usual, I'll present my normal workflow, that involves calling `pdflatex` two times to get references right, one call to `makeindex` and finally, a last call to `pdflatex`. Though there's no need of calling `pdflatex` two times in the beginning, I'll keep that as a good practice from my side.

By running `arara helloindex` or `helloindex.tex` in the terminal, we will obtain the same output from Code 19. The execution order is defined

Code 17 • `makeindex.yaml`, second attempt.

```yaml
!config
identifier: makeindex
name: MakeIndex
command: makeindex @{german} @{style} @{file}.idx
arguments:
- identifier: style
  flag: -s @{value}
- identifier: german
  flag: '@{value.toLowerCase() == "yes" || value.toLowerCase
      () == "true" || value.toLowerCase() == "on" ? "-g" : ""
       }'
```

Code 18 • `helloindex.tex`

```tex
% arara: pdflatex
% arara: pdflatex
% arara: makeindex
% arara: pdflatex
\documentclass{article}

\usepackage{makeidx}

\makeindex

\begin{document}

Hello world\index{Hello world}.

Goodbye world\index{Goodbye world}.

\printindex

\end{document}
```

by the directives order in the `.tex` file. If any command fails, arara halts at that position and nothing else is executed.

You might ask how arara knows if the command was successfully executed. The idea is quite simple: good programs like pdflatex make use of a concept known as exit status. In short, when a program had a normal execution, the exit status is zero. Other values are returned when an abnormal execution happened. When pdflatex successfully compiles a .tex file, it returns zero, so arara intercepts this number. Again, it's a good practice to make command line applications return proper exit status according to the execution flow, but beware: you might find applications or shell commands that don't feature this control (in the worst case, the returned value is always zero).

Code 19 • Running helloindex.tex.

```
$ arara helloindex
  __ _ _ __ __ _ _ __ __ _
 / _` | '__/ _` | '__/ _` |
| (_| | | | (_| | | | (_| |
 \__,_|_| \__,_|_| \__,_|

Running PDFLaTeX... SUCCESS
Running PDFLaTeX... SUCCESS
Running MakeIndex... SUCCESS
Running PDFLaTeX... SUCCESS
```

According to the terminal output shown in Code 19, arara executed all the commands successfully. In Chapter 4 we will learn more about how arara deals with commands and how to get their outputs for a more detailed analysis.

Cleaning temporary files

After running arara helloindex successfully (Code 19), we now have as a result a new helloindex.pdf file, but also a lot of auxiliary files, as we can see in Code 20.

What if we write a new clean rule to remove all the auxiliary files? The idea is to use rm to remove each one of them. As mentioned in the beginning of the manual, some rules might be system-specific, so this one is a perfect example.

Code 20 • List of auxiliary files after running `arara helloindex`.

```
$ ls
helloindex.aux helloindex.ilg helloindex.log helloindex.tex
helloindex.idx helloindex.ind helloindex.pdf
```

Since we want our rule to be generic enough, it's now a good opportunity to introduce the use of the reserved directive key `files`, first seen in Section 3.3. This special key is a list that overrides the default `@{file}` value and replicates the directive for every element in the list. I'm sure this will be the easiest rule we've written so far. The `clean` rule is presented in Code 21.

Code 21 • `clean.yaml`

```
1 !config
2 identifier: clean
3 name: CleaningTool
4 command: rm -f @{file}
5 arguments: []
```

Note that the command `rm` has a `-f` flag. As mentioned before, commands return an exit status after their calls. If we try to remove a nonexistent file, `rm` will complain and return a value different than zero. This will make arara halt and print a big FAILURE on screen, since it is considered an abnormal execution. If we provide the `-f` flag, `rm` will not complain of a nonexistent file, so we won't be bothered for this trivial task.

Now we need to add the directive to our `helloindex.tex` file (Code 18). Of course, `clean` will be the last directive, since it will only be reachable if everything executed before was returned no errors. The new header of `helloindex.tex` is presented in Code 22.

The reserved directive key `files` has five elements, so the `clean` rule will be replicated five times with the orb tag `@{file}` being expanded to each element. If you wish, you can also evaluate the value through conditional expression, as we did before with the other rules. In my opinion, I don't think it's necessary for this particular rule.

Time to run `arara helloindex` again and see if our new `clean` rule works! Code 23 shows both arara execution and directory listing. We expect to

Code 22 • `helloindex.tex` with the new `clean` directive.

```
% arara: pdflatex
% arara: pdflatex
% arara: makeindex
% arara: pdflatex
% arara: clean: { files: [ helloindex.aux, helloindex.idx,
    helloindex.ilg, helloindex.ind, helloindex.log ] }
\documentclass{article}
...
```

find only our source `helloindex.tex` and the resulting `helloindex.pdf` file.

Code 23 • Running `helloindex.tex` with the new `clean` rule.

```
$ arara helloindex
  __ _ _ __ __ _ _ __ __ _
 / _` | '__/ _` | '__/ _` |
| (_| | | | (_| | | | (_| |
 \__,_|_| \__,_|_| \__,_|

Running PDFLaTeX... SUCCESS
Running PDFLaTeX... SUCCESS
Running MakeIndex... SUCCESS
Running PDFLaTeX... SUCCESS
Running CleaningTool... SUCCESS
Running CleaningTool... SUCCESS
Running CleaningTool... SUCCESS
Running CleaningTool... SUCCESS
Running CleaningTool... SUCCESS
$ ls
helloindex.pdf helloindex.tex
```

Great, the `clean` rule works like a charm! A friendly note: if you are in Windows, replacing `rm` by the equivalent `del` won't probably work. Commands like `del` must be called in the form `cmd /c del`. Make sure to exhaustively test your rules before putting them into production. Check Chapter 4 to learn more about tracking the arara execution.

Chapter 4

When something goes wrong

Don't panic!

From The Hitchhiker's
Guide to the Galaxy

One of arara's goals is to reduce the verbosity of commands. Though the extensive output might contain relevant information about the execution process, in most of the cases it is simply to much stuff going on for us to follow. Besides, commands like `pdflatex` generate a proper `.log` file for us to check how things went. arara's minimalist approach informs us about the execution status: SUCCESS or FAILURE. When things go terribly wrong, we need to rely on more than this status. We should ask arara to keep track of the execution plan for us.

4.1 arara messages

arara messages are the first type of feedback provided by arara. These messages are basically related to rules and directives. Bad syntax, nonexisting rules, malformed directives, wrong expansion, arara tries to tell you what went wrong. Those messages are usually associated with errors. I tried to include useful messages, like telling in which directive and line an error ocurred, or that a certain rule does not exist or has an incorrect format. arara also checks if a command is valid. If you try to call a rule that executes a nonexisting `makefoo` command, arara will complain about it.

4.2 Logging

Another way of looking for an abnormal behaviour is to read the proper `.log` file. Unfortunately, not every command emits a report of its execution and, even if the command generates a `.log` file, multiple runs would overwrite the previous reports and we would have only the last call. arara provides a more consistent way of monitoring commands and their own behaviour through a global `.log` file that holds every single bit of information. You can enable the logging feature by adding either the `--log` or `-l` flags to the `arara` application.

Before we continue, I need to explain about standard streams, since they constitute an important part of the generated `.log` file by arara. Wikipedia has a nice definition of them:

> "In computer programming, standard streams are preconnected input and output channels between a computer program and its environment (typically a text terminal) when it begins execution. The three I/O connections are called standard input (`stdin`), standard output (`stdout`) and standard error (`stderr`)."

Basically, the operating system provides two streams directed to display data: `stdout` and `stderr`. Usually, the first stream is used by a program to write its output data, while the second one is typically used to output error messages or diagnostics. Of course, the decision of what output stream to use is up to the program author.

When arara traces a command execution, it logs both `stdout` and `stderr`. The log entry for `stdout` is `Standard output logging` while `stderr` is referenced by `Standard error logging`. Again, an output to `stderr` does not necessarily mean that an error was found in the code, while an output to `stdout` does not necessarily mean that everything ran flawlessly. It's just a naming convention, as the program author decides how to handle the messages flow. That's why arara logs them both. Read the `stdout` and `stderr` log entries carefully. A excerpt of the resulting `arara.log` from `arara helloindex --log` is show in Code 24 – several lines were removed in order to leave only the more important parts.

The arara log is useful for keeping track of the execution flow as well as providing feedback on how both rules and directives are being expanded. The log file contains information about the directive extraction and parsing, rules checking and expansion, deployment of tasks and execution of commands. The arara messages are also logged.

Code 24 • `arara.log` from `arara helloindex --log`.

```
09 Abr 2012 11:27:58.400 INFO Arara - Welcome to Arara!
09 Abr 2012 11:27:58.406 INFO Arara - Processing file helloindex.tex,
    please wait.
09 Abr 2012 11:27:58.413 INFO DirectiveExtractor - Reading directives from
     helloindex.tex.
09 Abr 2012 11:27:58.413 TRACE DirectiveExtractor - Directive found in
    line 1 with pdflatex.
...
09 Abr 2012 11:27:58.509 INFO DirectiveParser - Parsing directives.
09 Abr 2012 11:27:58.536 INFO TaskDeployer - Deploying tasks into commands
    .
09 Abr 2012 11:27:58.703 INFO CommandTrigger - Ready to run commands.
09 Abr 2012 11:27:58.704 INFO CommandTrigger - Running PDFLaTeX.
09 Abr 2012 11:27:58.704 TRACE CommandTrigger - Command: pdflatex -
    interaction=nonstopmode helloindex.tex
09 Abr 2012 11:27:59.435 TRACE CommandTrigger - Standard error logging:
09 Abr 2012 11:27:59.435 TRACE CommandTrigger - Standard output logging:
    This is pdfTeX, Version 3.1415926-2.3-1.40.12 (TeX Live 2011)
...
Output written on helloindex.pdf (1 page, 12587 bytes).
Transcript written on helloindex.log.
09 Abr 2012 11:27:59.435 INFO CommandTrigger - PDFLaTeX was successfully
    executed.
09 Abr 2012 11:27:59.655 INFO CommandTrigger - Running MakeIndex.
09 Abr 2012 11:27:59.655 TRACE CommandTrigger - Command: makeindex
    helloindex.idx
09 Abr 2012 11:27:59.807 TRACE CommandTrigger - Standard error logging:
    This is makeindex, version 2.15 [TeX Live 2011] (kpathsea + Thai
    support).
...
Generating output file helloindex.ind..done (9 lines written, 0 warnings).
Output written in helloindex.ind.
Transcript written in helloindex.ilg.
09 Abr 2012 11:27:59.807 TRACE CommandTrigger - Standard output logging:
09 Abr 2012 11:27:59.807 INFO CommandTrigger - MakeIndex was successfully
    executed.
...
09 Abr 2012 11:28:00.132 INFO CommandTrigger - All commands were
    successfully executed.
09 Abr 2012 11:28:00.132 INFO Arara - Done.
```

If by any chance your code is not working, try to run `arara` with the logging feature enabled. It might take a while for you to digest the log entries, but I'm sure you will be able to track every single step of arara's execution and fix the offending line in your code.

Chapter 5

Best practices

Snakes! Why did it have to be snakes?

Indiana Jones,
Raiders of the Lost Ark (1981)

The following list contains some hints on best practices when using arara. I tried my best to name a few situations and annoyances that you might encounter, but the list is far from being complete and accurate. Feel free to establish your own practices. After all, arara depends on the user, and not the other way around.

5.1 Plain rules

Use a text editor with support to `.yaml` files
In my humble opinion, YAML is a great format for expressing arara rules, but you might encounter problems if the `.yaml` file is not well-formed. Please follow the rule format presented in Section 3.1 and use a text editor with proper support to the YAML format. Personally, I use Vim for editing arara rules.

Use only lowercase letters when defining identifiers for rules
Avoid at all costs uppercase letters, digits, spaces, punctuation or other symbols when defining the `identifier` key for arara rules.

Prefer to enclose orb tags with single quotes
Although you can also enclose orb tags with double quotes, I suggest

you to stick with single quotes. Use double quotes inside the orb tag for possible evaluations.

If the key value only contains an orb tag, enclose it

Compare the `flag` key of the two arguments from the `makeindex` rule (Code 17). When there's only the orb tag as value or if the orb tag comes first in the value, please enclose the whole value with single quotes, like `'@{value}'` or `'@{value} --flag'`. arara tries to resolve the value type, but sometimes orb tags can mislead the extractor. If you want to play it safe, enclose the value with single quotes.

Don't use reserved keywords as identifiers

arara has a few reserved keywords: `file`, `files`, `value` and `arara`. Don't use them as identifiers, otherwise name clashes will make arara's behaviour unpredictable and mess with the document workflow.

5.2 Directives

If an argument value has spaces, enclose it with quotes

Again, try to avoid at all costs values with spaces, but if you really need them, enclose the value with single quotes. Beware: commands might require you to enclose values with spaces with double quotes! If you try to run the directive `clean: { files: [ 'my doc.aux' ] }`, the command will be expanded to `rm -f my doc.aux` which is wrong! Two files will be removed: `my` and `doc.aux`. The solution is to use double quotes inside the value surround by single quotes, so a call to the directive `clean: { files: [ '"my␣doc.aux"' ] }` will be expanded to `rm -f "my␣doc.aux"` which is correct. Another example is the `makeindex` directive. If you have a style named `my style.ist`, you can call it by running `makeindex: { style: '"my␣style"' }` and the command will be correctly expanded.

If you want to make sure that both rules and directives are being mapped and expanded correctly, enable the logging option with the `--log` flag and verify the output. All expansions are logged.

Chapter 6

Predefined rules

On the next pages you will find some predefined styles which can be used out of the box.

However it's important to save the rules in the folder `ARARA_HOME/rules/plain` (see section 3.1).

The description of the optional arguments contains the flag argument `yes`. Please note you can also use `on` or `true` in capitals or not. That means values like `TrUe` or `YES` are also welcome.

If you have some improvements or new rules please inform me.

The rules `latex`, `pdflatex`, `xelatex` and `lualatex` provide the option `write18`. This option doesn't work out of the box because the *orb tag* isn't set in the command line. If you use MikTeX you have to change the *orb tag* of the command line from `@{shell}` to `@{write18}` to work with the optional argument `write18`. This behavior depends on the facht that the rules were written for using with TeX Live.

arara rule

latex

The rule calls the compiler `latex`. The rule doesn't have a mandatory argument. Inside your `.tex` file you can use:

```
1 % arara: latex
```

Options

`shell`	If you say `shell:yes` you compile with `--shell-escape` otherwise with `--no-shell-escape`
`write18`	If you say `write:yes` you compile with `--enable-write18` otherwise with `--disable-write18`
`synctex`	If you say `synctex:yes` you compile with `--synctex=1` otherwise with `--synctex=0`
`draft`	If you say `draft:yes` you compile with `-draftmode` otherwise with `--draftmode=off`
`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

pdflatex

The rule calls the compiler `pdflatex`. The rule doesn't have a mandatory argument. Inside your `.tex` file you can use:

```
1  % arara: pdflatex
```

Options

`shell`	If you say `shell:yes` you compile with `--shell-escape` otherwise with `--no-shell-escape`
`write18`	If you say `write:yes` you compile with `--enable-write18` otherwise with `--disable-write18`
`synctex`	If you say `synctex:yes` you compile with `--synctex=1` otherwise with `--synctex=0`
`draft`	If you say `draft:yes` you compile with `-draftmode` otherwise with `--draftmode=off`
`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

lualatex

The rule calls the compiler `lualatex`. The rule doesn't have a mandatory argument. Inside your `.tex` file you can use:

```
1 % arara: lualatex
```

Options

`shell`	If you say `shell:yes` you compile with `--shell-escape` otherwise with `--no-shell-escape`
`write18`	If you say `write:yes` you compile with `--enable-write18` otherwise with `--disable-write18`
`synctex`	If you say `synctex:yes` you compile with `--synctex=1` otherwise with `--synctex=0`
`draft`	If you say `draft:yes` you compile with `-draftmode` otherwise with `--draftmode=off`
`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

xelatex

The rule calls the compiler `xelatex`. The rule doesn't have a mandatory argument. Inside your `.tex` file you can use:

```
% arara: xelatex
```

Options

`shell`	If you say `shell:yes` you compile with `--shell-escape` otherwise with `--no-shell-escape`
`write18`	If you say `write:yes` you compile with `--enable-write18` otherwise with `--disable-write18`
`synctex`	If you say `synctex:yes` you compile with `--synctex=1` otherwise with `--synctex=0`
`draft`	If you say `draft:yes` you compile with `-draftmode` otherwise with `--draftmode=off`
`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

biber

The rule calls the bibliography compiler `biber`. The compiler is recommended by `biblatex` and so implemented by arara. The rule doesn't have a mandatory argument. Inside your `.tex` file you can use:

```
% arara: biber
```

Options

`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

bibtex

The rule calls the standard bibliography compiler `bibtex`. The rule doesn't have a mandatory argument. Inside your `.tex` file you can use:

```
1 % arara: biber
```

Options

`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

makeglossaries

The rule calls the perl script `makeglossaries` provided by the package `glossaries` The rule doesn't have a mandatory argument. Inside your `.tex` file you can use:

```
1 % arara: makeglossaries
```

Options

`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

makeindex

The rule calls the standard index processor of LaTeX. The rule doesn't have a mandatory argument. Inside your `.tex` file you can use:

```
% arara: makeindex
```

Options

`style`	You can specify a style file which will be used by `makeindex`.
`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

nomencl

The rule calls the correct `makeindex` run which is required by the package `nomencl`. The rule doesn't have a mandatory argument. Inside your `.tex` file you can use:

```
% arara: nomencl
```

Options

`log`	If you say `log:yes` the compilation by `makeindex` will create a log file named `mynomencl.nlg`
`stylefile`	If you want to use your own style file for the compilation by `makeindex` you can give the name of the file as argument to the option.
`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

make

The rule calls the Unix command `make`. The argument of make can be handled as mandatory because it depends on you `Makefile`.

```
1 % arara: make
```

Options

`task`	Specify the task of `make`

arara rule

dvips

The rule calls the converting command `dvips`.

```
1 % arara: make
```

Options

`outputfile`	Declare the name of the output file. By default the name which is save in the *orb tag* `file` is used.
`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

ps2pdf

The rule calls the converting command `ps2pdf`.

```
% arara: make
```

Options

`outputfile`	Declare the name of the output file. By default the name which is save in the *orb tag* `file` is used.
`expandoptions`	Every value of this options will be passed to the compilation run direct. So you can expand the option list.

arara rule

removehelpfiles

The rule works only with a Unix System. Instead of explaining the options I will show the definition of the rule.

```
!config
identifier: removehelpfiles
name: Removing-Help-Files
command: 'rm -f @{remove}'
arguments:
  - identifier: remove
    flag: '@{value}'
```

arara rule

cleanhelpfiles

The rule works only with a Unix System. Instead of explaining the options I will show the definition of the rule.

```
!config
identifier: cleanhelpfiles
name: Removing-Help-Files
command: find -type f ! -iname ''*.pdf'' ! -iname ''*.
    tex'' ! -iname ''*.sty''
                  ! -iname ''*.def'' ! -iname ''*.bst''
                      -exec rm '{}' +
arguments:
  - identifier: keep
    flag: '@{value}'
```

Part II

Chapter 7

▼ Writing compiled rules