The cool TEX automation tool

User Manual

Paulo R. M. Cereda
cereda@users.sf.net

Marco Daniel
marco.daniel@mada-nada.de

Brent Longborough
brent@longborough.org

Version 3.0

Prologue

Moral of the story: never read the documentation, bad things happen.

David Carlisle

When I released the very first version of arara on a Friday 13th, April 2012, I never thought the tool would receive so many positive comments and feedback. To be honest, since arara was written for helping me with my own personal LaTeX projects, I really doubted if the tool could be of service to anyone else. And, to my surprise, it seems arara did good. To a lot of people around the TeX world.

I never intended to release the tool 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, although with different approaches. The reason I decided to make arara publicly available is quite simple: I wanted to somehow contribute to the TeX community, and I wanted to give my best to make such community even more awesome.

As time goes by, I'm quite satisfied with the current state of arara – this is our 3rd major release. We have reached a very mature code and a great team of developers, translators and testers. Since version 1.0, the code evolved a lot – new features, lots of bug fixes, improvements – thanks to all the feedback I received. In my humble opinion, that's how any project should evolve: based on what our users expect and want to achieve. I'm proud to see arara being 100% community-driven, it's a big achievement for a project with less than one year old.

First of all, I'd like to thank some friends of mine that really made arara possible: Alan Munn, for providing great ideas and suggestions to the manual; Andrew Stacey, for heavily testing arara, providing great user cases, and for suggesting improvements to the program; Brent Longborough, a member of the core team, for providing great suggestions and ideas to the program logic, writing rules, testing the code and also for working with the Portuguese and Turkish translations; Clemens Niederberger, for testing arara, and also writing a great tutorial about it in his blog on chemistry and LaTeX; David Carlisle, for reminding me to work on arara, and also encouraging me to write answers about it in our TeX community; Enrico Gregorio, for reviewing the original manual, testing arara, providing great ideas and suggestions to the manual and to the program itself, and for working with the Italian translation; Francesco Endrici, for providing the very first arara rule outside our core team; Harish Kumar, for being a heavy arara user and integrating it with WinEdt and Inlage; İlhan Polat for working with Brent in the Turkish translation; Joseph Wright, for testing it, providing contributed code for Linux and Mac installations, and also blogging about arara in his personal blog; Gonzalo Medina, for providing the Spanish translation; Mikaël Maunier, for providing the French translation; Marco Daniel, one of core team members, for heavily testing arara, suggesting enhancements to the manual and to the program itself, providing lots of contributed rules for common tasks, and also for the German version; Patrick Gundlach, for advertising arara in the official Twitter channel of Dante – the German TeX User Group; Sergey Ulyanov, for providing the Russian translation and contributed rules; Stefan Kottwitz, for encouraging me to write an article about arara, published in the LaTeX Community forum, and also tweeting about it. Thank you very much. I'm sorry if I forgot to mention somebody, I really have so much people to thank and my memory happens to be very short.

That said, I still believe that the warning featured in the first version of this manual still applies: Hic Sunt Dracones. Though the code really evolved from the first commit I made, arara is far from being bug-free. And you will learn 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 greatest features is the freedom it offers. But as you know, freedom always comes at a cost. Please, don't send us angry letters – or e-mails, perhaps – if something bad happen.

Feedback is surely welcome for me to improve this humble tool, just write an e-mail to me or any other member of the team and we will 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, submitting bugs, sending pull requests or even translating it to your language. If you want to support the 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
on behalf of the arara *team*

Special thanks

Alan Munn	Andrew Stacey	Brent Longborough
Clemens Niederberger	David Carlisle	Enrico Gregorio
Francesco Endrici	Harish Kumar	İlhan Polat
Joseph Wright	Gonzalo Medina	Mikaël Maunier
Marco Daniel	Patrick Gundlach	Sergey Ulyanov
Stefan Kottwitz		

arara also makes use of some specific opensource Java projects and libraries in order to properly work. I would like to thank the following projects and their respective developers:

1. Apache Commons, a project from the Apache Foundation focused on all aspects of reusable Java components. **arara** uses three of the Commons libraries: CLI, which provides a command line arguments parser, Collections, a library which extends the Java Collections Framework, and Exec, an API for dealing with external process execution and environment management in Java.

2. Logback, a logging framework intended to be the successor to the popular log4j project. According to some benchmarks, it is faster and has a smaller footprint than all existing logging systems, sometimes by a wide margin.

3. SnakeYAML, a YAML parser and emitter for the Java programming language. YAML is a data serialization format designed for human readability and interaction with scripting languages. **arara** uses YAML as the rule format.

4. SLF4J, a simple facade or abstraction for various logging frameworks, allowing the end user to plug in the desired logging framework at deployment time.

5. MVEL, a powerful expression language for Java-based applications. It provides a plethora of features and is suited for everything from the smallest property binding and extraction, to full blown scripts. **arara** relies on MVEL to provide the expansion mechanism for rules.

6. Apache Maven, a software project management and comprehension tool. Based on the concept of a project object model, Maven can manage a project's build, reporting and documentation from a central piece of information.

7. IzPack, a Java-based software installer builder that will run on any operating system coming with a Java Virtual Machine that is compliant with the Oracle JVM 1.5 or higher.

A special thanks goes to my great friend Antoine Neveux for encouraging me to try out the Apache Maven software project management. In the past, arara was released as a NetBeans project, which is based on Apache Ant, another great tool from the Apache Foundation. Although I'm really fine with Ant, thanks to Maven, now it is way easier to build and to maintain the code. And it's always nice to learn another tool.

And at last but not least, I want to thank you, dear reader and potential user, for giving arara a try. Do not despair if you don't succeed with arara at first; just try again. I'm sure you will find your way. This humble project is opensource and it will always be. Let the bird be your guide through the journey to the typographic land. Have a good read.

Release information

Version 3.0

new Localizated messages in English, Brazilian Portuguese, German, Italian, Spanish, French, Turkish and Russian.

fixed Improved error analysis for rules and directives.

new Friendly and very detailed messages instead of generic ones.

new An optional configuration file is now available in order to customize and enhance the application behaviour.

fixed Improved rule syntax, new keys added.

new Now rules are unified in a plain format. No more compiled rules.

new Rules can allow an arbitrary number of commands instead of just one.

new Built-in functions in the rule context to ease the writing process.

fixed Improved expansion mechanism.

Table 1: Lines of code for version 3.0.

Language	Files	Blank	Comment	Code
Java	25	847	2722	1659
XML	2	12	0	181
Sum	27	859	2722	1840

Version 2.0

new Added the `--timeout n` flag to allow setting a timeout for every task. If the timeout is reached before the task ends, arara will kill it and interrupt the processing. The n value is expressed in milliseconds.

fixed Fixed the `--verbose` flag to behave as a realtime output.

new There's no need of noninteractive commands anymore. arara can now handle user input through the `--verbose` tag. If the flag is not set and the command requires user interaction, the task execution is interrupted.

fixed Fixed the execution of some script-based system commands to ensure cross-platform compatibility.

new Added the `@{SystemUtils}` orb tag to provide specific operating system checks. The orb tag maps the `SystemUtils` class from the amazing Apache Commons Lang library and all of its methods and properties.

Table 2: Lines of code for version 2.0.

Language	Files	Blank	Comment	Code
Java	20	608	1642	848
XML	1	0	0	12
Sum	21	608	1642	860

Version 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.

Version 1.0

new First public release.

Table 3: Lines of code for version 1.0.1.

Language	Files	Blank	Comment	Code
Java	20	585	1671	804
XML	1	0	6	12
Sum	21	585	1677	816

Table 4: Lines of code for version 1.0.

Language	Files	Blank	Comment	Code
Java	20	524	1787	722
XML	1	0	6	12
Sum	21	524	1793	734

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.

To my cat Fubá, who loves birds.

Contents

List of Figures

List of Tables

List of Codes

Part I

The application

Chapter 1

Introduction

You can do such a lot with a Wompom, you can use every part of it too. For work or for pleasure, it's a triumph, it's a treasure, oh there's nothing that a Wompom cannot do.

Flanders & Swann

Hello there, welcome to arara! I'm glad you were not intimidated by the threatening message in the prologue. This chapter is actually 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! arara is a TEX automation tool based on rules and directives. It is, in some aspects, similar to other well-known tools like `latexmk` [2] and `rubber` [4]. The key difference might be the fact that arara aims at explicit instructions in the source code in order to determine what to do instead of relying on other resources, such as log file analysis. It's a different approach for an automation tool, and we have both advantages and disadvantages of such decision. Let's talk about disadvantages first.

Since we need to explicitly tell arara what we want it to do, it might not be intuitive for casual users. Tools like `latexmk` and `rubber` rely on a analysis scheme in which the document is generated with a simple call to `latexmk mydoc.tex` or `rubber --pdf mydoc.tex`, while a similar call to

`arara mydoc.tex` does absolutely nothing; it's not wrong, it's by design: arara needs to know what you want. We do this by adding a directive in our `.tex` file, as shown in line 1 of Code 1. Don't worry with the terms now, we will come back to the concepts later on in this manual, in Chapter 5.

Code 1: `mydoc.tex`

```
1 % arara: pdflatex
2 \documentclass{article}
3
4 \begin{document}
5 Hello world.
6 \end{document}
```

When we add a directive in our source code, we are explicitly telling arara what we want it to do, but I'm afraid that's not sufficient. So far, arara knows *what* to do, but now it needs to know *how* the task should be done. Then, for every directive, we need to have an associated rule. In other words, if we want arara to run `pdflatex` on `mydoc.tex`, we need to have a set of instructions which tells our tool how to run that specific application. Although the core team provides a lot of rules shipped with arara out of the box, with the possibility of extending the set by adding more rules, some users might find this decision rather annoying, since other tools have most of their rules hardcoded, making the automation process even more transparent.

Now, let's talk about some advantages. In my humble opinion, since arara doesn't rely on a specific automation or compilation scheme, it becomes more extensible. The use of directives in the source code make the automation steps more fluent, which allows the specification of complex workflows very easily. Maybe arara's verbosity on automation steps might not be suitable for small documents, but the tool really shines when you have a document which needs full control of the automation process.

Another advantage that comes to my mind right now is the fact that directives and rules can be parametrized. In other words, you can create conditional branches, execution workflows based on parameters, flags, and so on, by simply providing a parameter in a directive. Besides, arara also provides a lot of helper functions in order to enhance rules; for example, you can have a rule which executes a certain command when in Windows, and a different one when in Unix.

The rules are written in a human-readable format. The reason for this decision came as an attempt to simplify the life of many casual users which

are not versed into programming. Sadly, writing complex XML mappings or even deliberately injecting code into an application is not a trivial task, so we opted for an easy way of declaring the set of instructions that tells arara how to do a task. We will discuss about the format later on, in Section 5.1.

Now that arara was properly introduced, let me explain the meaning of the name. *Arara* is the Brazilian name of a macaw bird (Figure 1.1). Have you ever watched *Rio: the movie*, produced by Blue Sky Studios? The protagonist is a blue arara. The word *arara* comes from the Tupian word *a'rara*, which means *big bird* [5].

Figure 1.1: A lovely photo of an arara.

Lovely bird, isn't it? Now, you are probably wondering why 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? And there is also another motivation of the name *arara*: the chatroom residents of TEX.sx – including myself – are fans of palindromes, especially palindromic numbers. As you can already tell, *arara* is a palindrome.

1.2 How does it work?

Now that we know what arara is, let's take a look on how the tool actually works. The whole idea is pretty straightforward, but some concepts might be confusing at first. Do not despair, we will come back to them later on in the manual, in Chapter 5.

First of all, we need to add at least one instruction in the source code to tell arara what to do. This instruction is named *directive* and it will be parsed during the preparation phase. By default, an arara directive is

defined in a line of its own, started with a comment, followed by the word `arara:` and the name of the task. Code 1 has one directive, referencing `pdflatex`. It's important to observe that `pdflatex` is not the command to be executed, but the name of the rule associated with that directive.

Once arara finds a directive, it will look for the associated *rule*. In our example, it will look for a rule named `pdflatex` which will evidently run the `pdflatex` command line application. The rule is analyzed, all possible parameters are defined, the command line call is built and then it goes to a queue of commands to be executed.

After extracting all directives from a source code and mapping each one of them to their respective rules, arara then executes the queue of commands. The execution chain requires that the command i was successfully executed to then proceed to the command $i + 1$, and so forth. This is also by design: arara will halt the execution if any of the commands in the queue had raised an error. If we run arara on `mydoc.tex` – we can also run `arara mydoc` too, we will discuss this later on – presented in Code 1, we get the output presented in Code 2.

Code 2: Running arara on `mydoc.tex`.

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

Running PDFLaTeX... SUCCESS
```

That is pretty much how arara works: directives in the source code are mapped to rules, which are converted to commands and added to a queue. The queue is then executed and the status is reported. We will cover more details about the expansion process later on in the manual. In short, we teach arara to do a task by providing a rule, and tell it to execute it via directives in the source code.

1.3 Features

To name a few features I like in arara, I'd mention the ability to write rules in a human-readable format called YAML, which rhymes with the word *camel*. YAML is actually a recursive acronym for *YAML Ain't Markup*

Language, and it's known as a human friendly data serialization standard for all programming languages [7]. So far, I think this format is quite suitable to write rules, specially if you want to avoid the need of writing complicated XML mappings or even injecting code directly into the application.

Another feature worth mentioning is the fact that arara is platform independent. The application was written in Java, so arara runs on top of a Java virtual machine, available on all the major operating systems – in some cases, you might need to install the proper virtual machine. We tried very hard to keep both code and libraries compatible with older virtual machines or from other vendors. Currently, arara is known to run on Oracle's Java 5, 6 and 7, and OpenJDK 6 and 7. In Chapter 3, there are instructions on how to build arara from sources. Even if you use multiple operating systems, arara should behave the same, including the rules. There are helper functions available in order to provide support for system-specific rules based on the underlying operating system, presented in Section 11.1.

From version 3.0 on, arara can now display localized messages. The default language is set to English, but the user can receive feedback from the execution process and logging in other languages as well, such as Brazilian Portuguese, German, Italian, French, Spanish, Russian and Turkish. There's also a way to redefine the default language by adding an entry in the configuration file, discussed later on in Section 6.2.

Speaking of which, arara has now an optional configuration file in which we can add rule paths, set the default language and define custom extensions and directive patterns, located in the user home directory. That way, we can extend arara's behaviour to deal with other extensions, such as `.c` files, and use the tool with other formats. We will come back on this subject later on in Chapter 6.

arara is also easily integrated with other TeX integrated development environment, such as TeXworks [3], an environment for authoring TeX documents shipped with both TeX Live and MiKTeX. Chapter 4 covers the integration of arara with several environments.

1.4 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, create glossaries, call `pdfcrop`, move files, run METAPOST or METAFONT, and so forth. You can easily come up with your own rules.

There's an article available in the LaTeX community which describes the integration of `gnuplot` and **arara** [1]. This article was submitted as an entry to a contest organized by Stefan Kottwitz. It might be worth a read.

Let's see a few examples. Code 3 contains the workflow I used for another 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 3: `article.tex`

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

Code 4 contains another 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 4: `manual.tex`

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

And of course, the **arara** user manual is also compiled with `arara`. You can take a look in the source code and check the header. By the way, note that I had to use a trick to avoid `arara` to read the example directives in this manual. As we will see later, **arara** reads directives everywhere. Actually, I could have changed the directive pattern for `.tex` files through the configuration file, but that's another story.

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. My friend Joseph Wright

wrote a great article about **arara** in his personal blog, it's really worth a read [6].

I really hope you like my humble contribution to the TEX community. Let **arara** enhance your TEX experience, it will help you when you'll need it the most. Enjoy the manual.

References

[1] Paulo Roberto Massa Cereda. *Fun with* `gnuplot` *and* `arara`. This article was submitted to the LATEX and Graphics contest organized by the LATEX community. 2012. URL: http://latex-community.org/know-how/435-gnuplot-arara (cit. on p. 8).

[2] John Collins. *Latexmk*. 2001. URL: http://www.phys.psu.edu/~collins/latexmk/ (cit. on p. 3).

[3] Jonathan Kew, Stefan Löffler, and Charlie Sharpsteen. *TEXworks: lowering the entry barrier to the TEX world*. 2009. URL: http://www.tug.org/texworks/ (cit. on p. 7).

[4] *Rubber*. The tool was originally developed by Emmanuel Beffara but the development largely ceased after 2007. The current team was formed to help keep the tool up to date. 2009. URL: https://launchpad.net/rubber (cit. on p. 3).

[5] *Tupi – Portuguese Dictionary*. URL: http://www.redebrasileira.com/tupi/vocabulario/a.asp (cit. on p. 5).

[6] Joseph Wright. **arara**: *making LATEX files your way*. 2012. URL: http://www.texdev.net/2012/04/24/arara-making-latex-files-your-way/ (cit. on p. 9).

[7] *YAML*. 2001. URL: http://www.yaml.org/ (cit. on p. 7).

Chapter 2

Installation

Adjust `\hsize`*: old man Fermat couldn't.*

Enrico Gregorio

Spledid, so you decided to give arara a try? This chapter will cover the installation procedure. We basically have two methods of installing arara: the first one is through a cross-platform installer, which is of course the recommended method; the second one is a manual deployment, with the provided `.jar` file – a self-contained, batteries-included executable Java archive file. If you have a recent TeX Live distribution, good news: arara is already available in your system!

2.1 Prerequisites

I know I've mentioned this before in Section 1.3 and, at the risk of being repetitive, there we go again: arara is written in Java and thus depends on a virtual machine in the underlying operating system. If you use a Mac or even a fairly recent Linux distribution, I have good news for you: it's mostly certain that you already have a Java virtual machine installed.

It's very easy to check if you have a Java virtual machine installed: try running `java -version` in the terminal (bash, command prompt, you name it) and see if you get an output similar to the one provided in Code 5.

If the output goes along the lines of `java: command not found`, I'm afraid you don't have a Java virtual machine installed in your operating system. Since the virtual machine is a prerequisite for arara to run, you can install one via your favorite package manager or manually install it from

Code 5: 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)
```

the binaries available in the official Java website. Make sure to download the correct version for your operating system. The installation procedure is very straightforward. If you get stuck, take a look on the installation instructions.

It's important to mention that arara runs also with the Java virtual machine from the OpenJDK project [7], which is already available in most of the recent Linux distributions – actually the output from Code 5 shows the OpenJDK version from my Fedora machine. Feel free to use the virtual machine you feel most comfortable with.

Speaking of virtual machines, arara requires at least Java 5 to run. Don't worry, it's quite easy to spot the Java version: just look at the second digit of the version string. For example, Code 5 outputs `1.6.0_24`, which means we have Java 6 installed.

2.2 Obtaining arara

Before proceeding, we need to choose the installation method. We have two options: the first option is the easiest one, which installs arara through a cross-platform installer; the second option is a manual deployment.

From version 3.0 on, arara is also available as part of the TeX Live distribution. If you have a recent TeX distro, it's almost certain that you already have arara; make sure to select it in the `tlmgr` application.

If we opt for the installer, go to the downloads section of the project repository and download `arara-3.0-installer.jar` for all operating systems or `arara-3.0-installer.exe` for Windows. Please note that the `.exe` version is only a wrapper which will launch `arara-3.0-installer.jar` under the hood. The installer also requires Java.

If we want to do things the complicated way, go to the downloads section of the project repository and download the `arara.jar` file, which is a self-contained, batteries-included executable Java archive file.

In case you want to build arara from source, please refer to Chapter 3 which will cover the whole process. Thanks to Apache Maven, the build

process is very easy.

2.3 Using the cross-platform installer

After downloading `arara-3.0-installer.jar` (or its `.exe` counterpart), it's now just a matter of running it. The installer is built with IzPack [4], an amazing tool for packaging applications on the Java platform. Of course the source is also available at the project repository. Personally, I suggest you to run the installer in privileged mode, but you can also run it in user mode – just keep in mind that some features might not work, like creating symbolic links or adding the application to the system path, which inevitably requires a privileged mode.

When running `arara-3.0-installer.jar` or its `.exe` wrapper on Windows by simply double-clicking it, the installer will automatically run in privileged mode. A general Unix-based installation can be triggered by the command presented in Code 6. There's also an alternative command presented in Code 7.

Code 6: Running the installer in a Unix-based system – method 1.

```
$ sudo java -jar arara-3.0-installer.jar
```

Code 7: Running the installer in a Unix-based system – method 2.

```
$ su -c 'java -jar arara-3.0-installer.jar'
```

Since Windows doesn't have a similar command to `su` or `sudo`, you need to open the command prompt as administrator and then run the command presented in Code 8. You can right-click the command prompt shortcut and select the "Run as administrator..." option.

Code 8: Running the installer in the Windows command prompt as administrator.

```
C:\> java -jar arara-3.0-installer.jar
```

The installation process will begin. Hopefully, the first screen of the installer will appear, which is the language selection (Figure 2.1). By the

way, if you called the installer through the command line, please do not close the terminal! It might end the all running processes, including our installer.

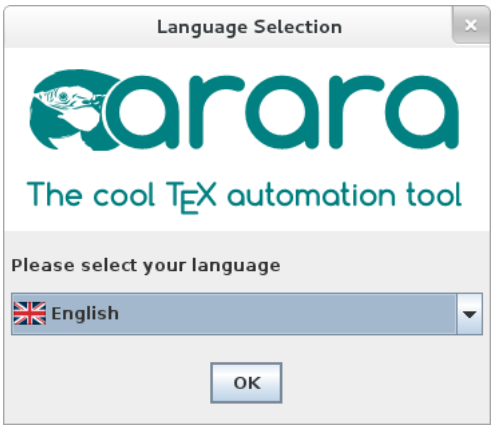

Figure 2.1: Language selection screen.

The installer currently supports six languages: English, German, French, Italian, Spanish, and Brazilian Portuguese. I plan to add more languages to the list in the near feature.

The next screen welcomes you to the installation (Figure 2.2). There's the application name, the current version, the team, and the project homepage. We can proceed by clicking the *Next* button. Note that you can quit the installer at any time by clicking the *Quit* button – please, don't do it; a kitten dies every time you abort the installation[1].

Moving on, the next screen shows the license agreement (Figure 2.4). arara is licensed under the New BSD License [6]. 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 [5], and has been vetted as an open source license by the Open Source Initiative [3]. The full license is also available in this document (page xi). You need to accept the terms of the license agreement before proceeding.

The next screen is probably the most important section of the installation: in here we will choose the packs we want to install (Figure 2.3). All packs are described in Table 2.1. Note that the grayed packs are required.

It's very important to mention that all these modifications in the operating system – the symbolic link creation for Unix or the addition to the

[1] Of course, this statement is just a joke. No animals were harmed, killed or severely wounded during the making of this user manual. After all, arara is environmentally friendly.

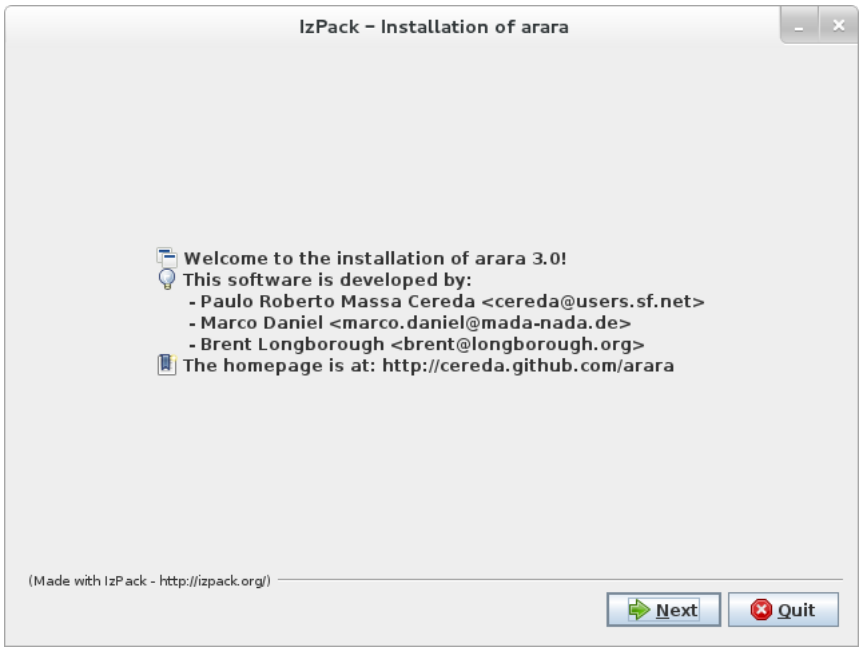

Figure 2.2: Welcome screen.

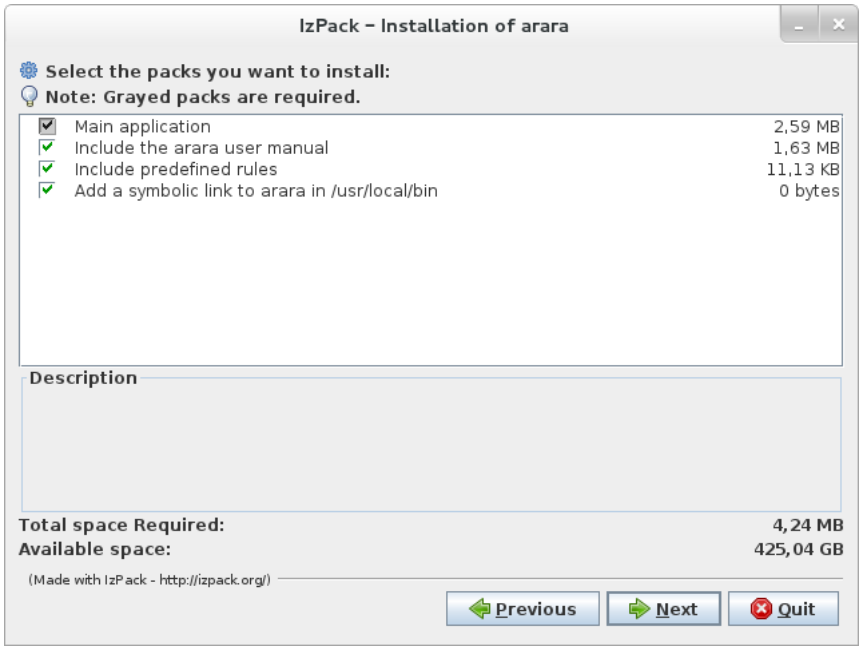

Figure 2.3: Packs screen.

Table 2.1: Available packs.

Pack name	OS	Description
Main application	All	This pack contains the core application. It also provides an `.exe` wrapper for Windows and a bash file for Unix.
Include the arara user manual	All	This pack installs this user manual into the `docs/` subdirectory of arara.
Include predefined rules	All	Of course, arara has a set of predefined rules for you to start with. If you prefer to write your own rules from scratch, do not select this pack.
Add a symbolic link to arara in `/usr/local/bin`	Unix	If you ran the installer in privileged mode, a symbolic link to arara can be created in the `/usr/local/bin` directory. There's no magic here, the installer uses the good old `ln` command.
Add arara to the system path	Windows	Like the Unix task, arara can also add itself to the system path. This feature is provided by a Windows script named Modify Path [1].

path for Windows – are safely removed when you run the arara uninstaller. We will talk about it later, in Section 2.6.

In the next screen, we will select the installation path (Figure 2.5). The installer will automatically set the default installation path according to the Table 2.2, but feel free to install arara in your favorite structure – even `/opt` or your home folder.

Table 2.2: Default installation paths.

OS	Default installation path
Windows	`C:\Program Files\arara`
Unix	`/usr/local/arara`

After selecting the installation path, the installer will then confirm the creation of the target directory (Figure 2.6). We simply click *OK* to accept

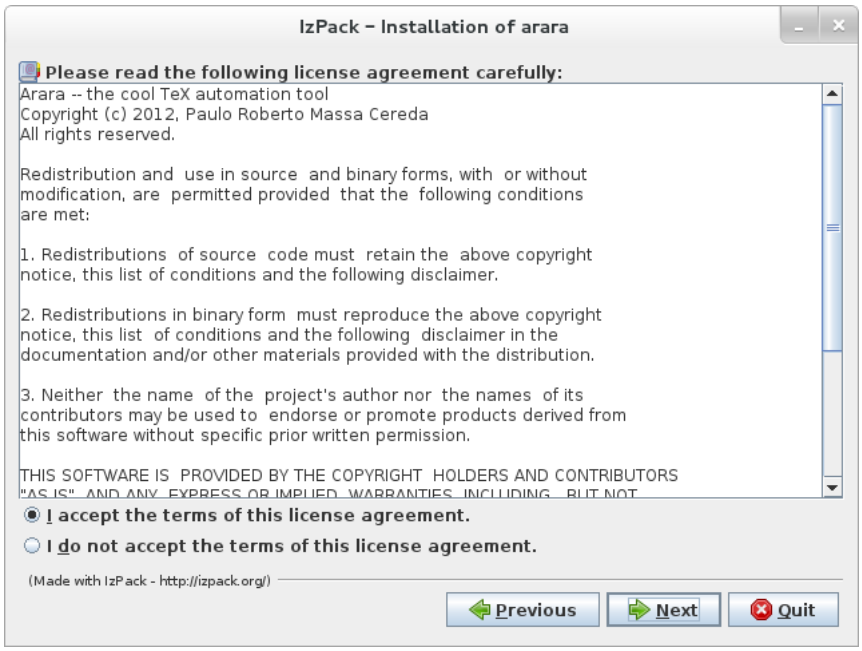

Figure 2.4: License agreement screen.

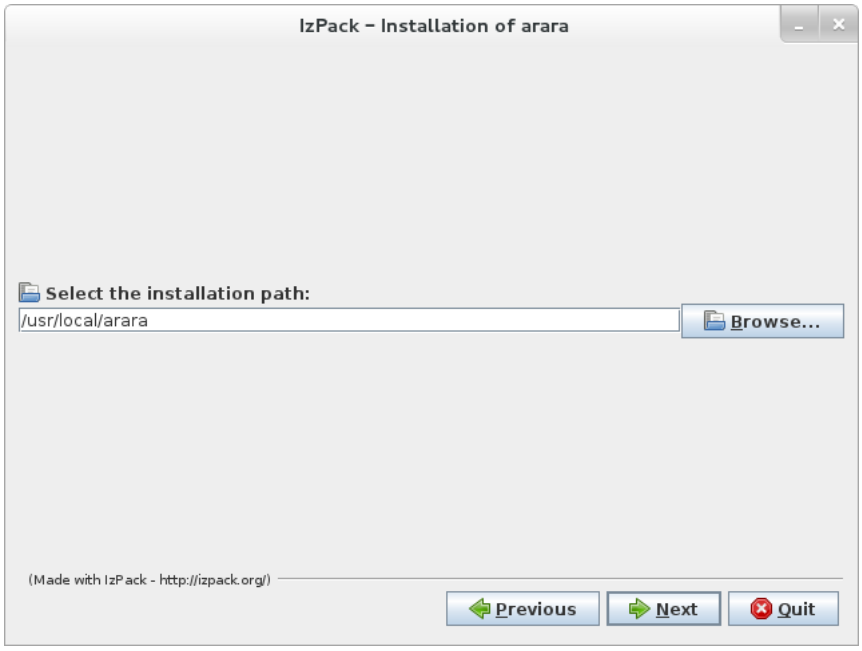

Figure 2.5: Installation path screen.

it. For convenience, the full installation path defined in the installation path screen (Figure 2.5) will be referred as `ARARA_HOME` from now on.

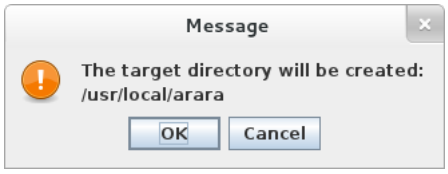

Figure 2.6: Target directory confirmation.

Now, just sit back and relax while arara is being installed (Figure 2.7). All selected packs will be installed accordingly. The post installation tasks – like creating the symbolic link or adding arara to the system path – are performed here as well. If the installation has completed successfully, we will reach the final screen of the installer congratulating us for installing arara (Figure 2.8).

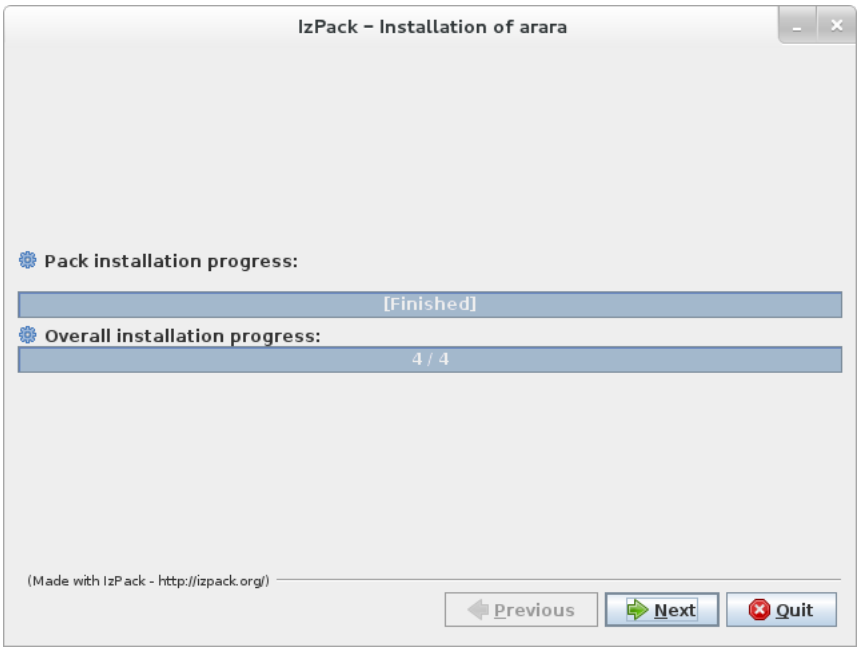

Figure 2.7: Progress screen.

The full installation scheme is presented in Figure 2.9. The directory structure is presented here as a whole; keep in mind that some parts will be omitted according to your operating system and pack selection. For example, the `etc/` subdirectory will only be installed if and only if you are

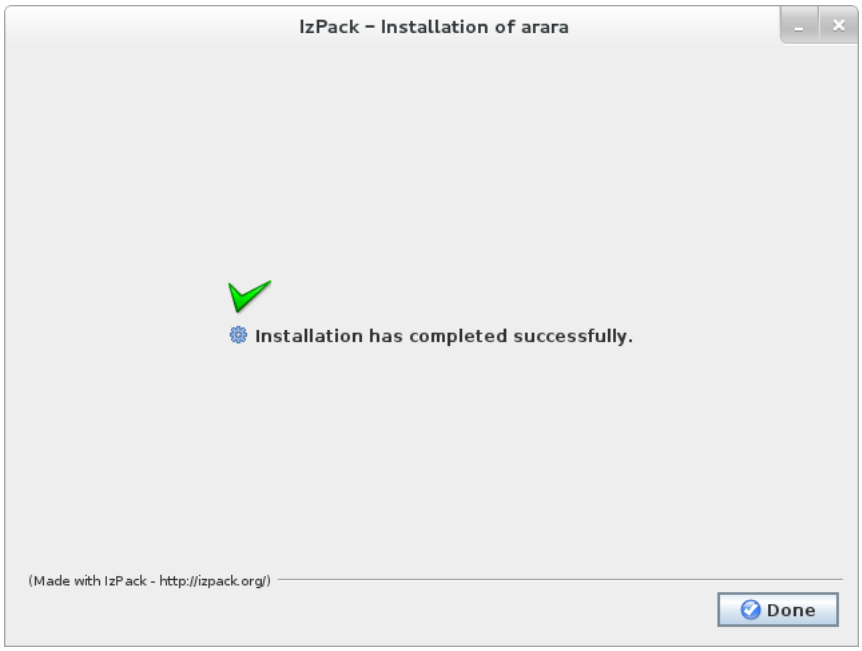

Figure 2.8: Final screen.

in Windows and the system path pack is selected. Other files are platform-specific, such as `arara.exe` for Windows and the `arara` bash file for Unix.

That's it, arara is installed in your operating system. If you opted for the symbolic link creation or the path addition, arara is already available in your terminal by simply typing `arara`. Have fun!

2.4 Manual installation

Thankfully, arara is also very easy to be manually deployed. First of all, we must create the application directory. Feel free to create this directory anywhere in your computer; it can be `C:\arara`, `/opt/arara` or another location of your choice. This procedure is similar to the installation path screen (Figure 2.5) from Section 2.3. Again, for convenience, the full installation path will be referred as `ARARA_HOME` from now on. Although it's not mandatory, try to avoid folders structures with spaces in the path. In any case, arara can handle such spaces.

After downloading `arara.jar` from the downloads section of the project repository, let's copy it to the `ARARA_HOME` directory we've created in the previous step. Since `arara.jar` is a self-contained, batteries-included executable Java archive file, arara is already installed.

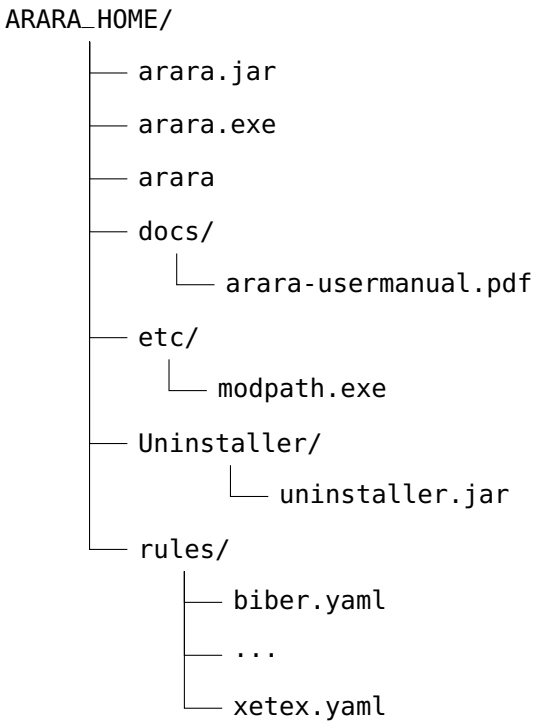

Figure 2.9: Installation scheme.

In order to run arara from a manual installation, we should open a terminal and run `java -jar $ARARA_HOME/arara.jar`, but that is far from being intuitive. To make our lives easier, we will create a shortcut for this command.

If you are deploying arara in Windows, there are two methods for creating a shortcut: the first method – the easiest – consists of downloading the `arara.exe` wrapper from the downloads section and copying it to the `ARARA_HOME` directory, in the same level of `arara.jar`. This `.exe` wrapper, provided by Launch4J [2], wraps `.jar` files in Windows native executables and allows to run them like a regular Windows program.

The second method for creating a shortcut in Windows is to provide a batch file which will call `java -jar $ARARA_HOME/arara.jar` for us. Create a file named `arara.bat` or `arara.cmd` inside the `ARARA_HOME` directory, in the same level of `arara.jar`, and add the content from Code 9.

After creating the batch file, add the full `ARARA_HOME` path to the system path. Unfortunately, this manual can't cover the path settings, since it's again a matter of personal taste. I'm sure you can find tutorials on how to add a directory to the system path.

Code 9: Creating a batch file for arara in Windows.

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

If you are deploying arara in Linux or Mac, we also need to create a shortcut to `java -jar $ARARA_HOME/arara.jar`. Create a file named `arara` inside the `ARARA_HOME` directory, in the same level of `arara.jar`, and add the content from Code 10.

Code 10: Creating a script for arara in Linux and Mac.

```
#!/bin/bash
# Example script of arara
# Installation and usage are described in the documentation
SOURCE="${BASH_SOURCE[0]}"
while [ -h "$SOURCE" ] ; do SOURCE="$(readlink "$SOURCE")";
    done
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && cd -P "$(
    dirname "$SOURCE" )" && pwd )"
java -jar "$DIR/arara.jar" "$@"
```

We now need to add execute permissions for our newly created script through `chmod +x arara`. The `arara` script can be invoked through path addition or symbolic link. I personally prefer to add `ARARA_HOME` to my user path, but a symbolic link creation seems way more robust – it's what the installer does. Anyway, it's up to you to decide which method you want to use. There's no need to use both.

Once we conclude the manual installation, it's time to check if arara is working properly. Try running `arara` in the terminal and see if you get the output shown in Code 11.

If the terminal doesn't display the arara logo and usage, please review the manual installation steps. Every step is important in order to make arara available in your system. You can also try the cross-platform installer. If you still have any doubts, feel free to contact us.

2.5 Updating arara

If there is a newer version of arara available in the downloads section of the project repository, simply download the `arara.jar` file and copy it to the `ARARA_HOME` directory, replacing the current one. No further steps are needed, the newer version is deployed. Try running `arara --version` in the terminaland see if the version shown in the output is equal to the one you have downloaded.

Anyway, for every version, arara has the proper cross-platform installer available for download in the project repository. You can always uninstall the old arara setup and install the new one. Please note that only major versions are released with the installer.

If you have arara through the TEX Live distribution, the update process is straightforward: simply open a terminal and run `tlmgr update arara` in order to update the application. This is of course the preferred method.

2.6 Uninstalling arara

If you want to uninstall arara, there are two methods available. If you installed arara through the cross-platform installer, I have good news for you: it's just a matter of running the uninstaller. Now, if arara was deployed through the manual installation, we might have to remove some links or path additions.

A general Unix-based uninstallation can be triggered by the command presented in Code 12. There's also an alternative command presented in Code 13.

Since Windows doesn't have a similar command to `su` or `sudo`, you need to open the command prompt as administrator and then run the command presented in Code 14. You can right-click the command prompt shortcut and select the "Run as administrator..." option.

The uninstallation process will begin. Hopefully, the first and only creen of the uninstaller will appear (Figure 2.10). By the way, if you called the uninstaller through the command line, please do not close the terminal! It might end the all running processes, including our uninstaller.

There's nothing much to see in the uninstaller. We have an option to force the deletion of the `ARARA_HOME` directory, but that's all. By clicking the *Uninstall* button, the uninstaller will remove the symbolic link or the path entry for arara from the operating system, if selected during the installation. Then it will erase the `ARARA_HOME` directory (Figure 2.11).

Code 11: Testing if arara is working properly.

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

arara 3.0 - The cool TeX automation tool
Copyright (c) 2012, Paulo Roberto Massa Cereda
All rights reserved.

usage: arara [file [--log] [--verbose] [--timeout N]
    [--language L] | --help | --version]

 -h,--help       print the help message
 -L,--language <arg> set the application language
 -l,--log        generate a log output
 -t,--timeout <arg> set the execution timeout (in milliseconds)
 -v,--verbose    print the command output
 -V,--version    print the application version
```

Code 12: Running the uninstaller in a Unix-based system – method 1.

```
$ sudo java -jar $ARARA_HOME/Uninstaller/uninstaller.jar
```

Code 13: Running the uninstaller in a Unix-based system – method 2.

```
$ su -c 'java -jar $ARARA_HOME/Uninstaller/uninstaller.jar'
```

Code 14: Running the uninstaller in the Windows command prompt as administrator.

```
C:\> java -jar $ARARA_HOME/Uninstaller/uninstaller.jar
```

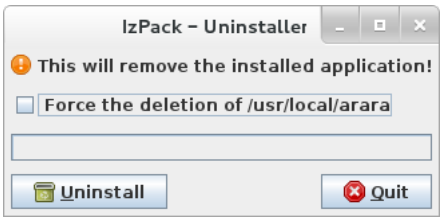

Figure 2.10: The uninstaller screen.

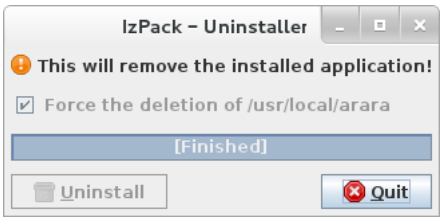

Figure 2.11: The uninstaller screen, after the execution.

Unfortunately, even if you force the deletion of the `ARARA_HOME` directory in Windows, the operating system can't remove the `Uninstaller` subdirectory because the uninstaller was being executed from there. But that's the only trace left. You can safely delete `ARARA_HOME` after running the uninstaller.

If arara was manually installed, we need to remove the symbolic link reference or the path entry, if any, then delete the `ARARA_HOME` directory. Don't leave any traces of arara in system directories or configuration files; a broken symbolic link or a wrong path entry might cause trouble in the future.

References

[1] Jared Breland. *Modify Path*. This tool in released under the GNU Lesser General Public License (LGPL), version 3. 2012. URL: http://legroom.net/software/modpath (cit. on p. 16).

[2] Grzegorz Kowal. *Launch4J, a cross-platform Java executable wrapper*. 2005. URL: http://launch4j.sourceforge.net/ (cit. on p. 20).

[3] Bruce Perens and Eric Steven Raymond. *Open Source Initiative*. Non-profit corporation with global scope formed to educate about and advocate for the benefits of open source and to build bridges among dif-

ferent constituencies in the open source community. 1998. URL: http://www.opensource.org/ (cit. on p. 14).

[4] Julien Ponge. *IzPack*. The project is developed by a community of benevolent contributors. 2001. URL: http://izpack.org/ (cit. on p. 13).

[5] Richard Stallman. *Free Software Foundation*. Nonprofit organization with a worldwide mission to promote computer user freedom and to defend the rights of all free software users. 1985. URL: http://www.fsf.org/ (cit. on p. 14).

[6] *The New BSD License*. URL: http://www.opensource.org/licenses/bsd-license.php (cit. on p. 14).

[7] *The OpenJDK Project*. 2006. URL: http://openjdk.java.net/ (cit. on p. 12).

Chapter 3

Building from sources

Knowledge brings fear.

From a *Futurama* episode

Although arara already features a self-contained, batteries-included executable Java archive file, an `.exe` wrapper, and a cross-platform installer, you can easily build it from sources. The only requirements are a working Java Development Kit [2] and the Apache Maven software project management [1]. The next sections will cover the entire process, from obtaining the sources to the build itself. Sadly, this manual doesn't cover Java and Maven deployments, so I kindly ask you to check their websites and read the available documentation.

3.1 Obtaining the sources

First of all, we need to get the source code, available in the project repository hosted on GitHub. We have two options on how to obtain the sources: either by clicking the Zip button in the project page and download a snapshot of the whole structure in an archive file, or by using `git` and clone the repository into our machine. The second option is easily done by executing the command presented in Code 15, provided of course that you have `git` installed.

Code 15: Cloning the project repository.

```
$ git clone git://github.com/cereda/arara.git
```

After cloning the project repository (Code 15), a new subdirectory named `arara` is created in the current directory with the project structure – the very same available in the project repository on GitHub. The application source code is inside `arara/arara`. Note that there are other source codes for the cross-platform installer and the `.exe` wrapper, as well as the predefined rules, each one in a subdirectory of its own.

If you opted for downloading the archive file, you'll have a file named `arara-master.zip` generated automatically by GitHub. Just extract the file somewhere in your computer and you'll end up with the very same project structure as the one available in the project repository.

3.2 Building arara

Inside the `arara/arara` directory, we have the most important file for building arara: a file named `pom.xml`. We now just need to call the `mvn` command with the proper target and relax while Maven takes care of the building process for us. First of all, let's take a look at some targets available in our `pom.xml` file:

`compile`
: This target compiles the source code, generating the Java bytecode.

`package`
: The `package` target is very similar to the `compile` one, but instead of only compiling the source code, it also packs the Java bytecode into an executable Java archive file without dependencies. The file will be available inside the `arara/arara/target` directory.

`assembly:assembly`
: This target is almost identical to the `package` one, but it also includes all the dependencies into a final Java archive file. The file will be available inside the `arara/arara/target` directory. This is of course our preferred target, since arara is shipped as a self-contained executable Java archive file.

`clean`
: The `clean` target removes all the generated Java bytecode and deployment traces, cleaning the project structure.

Now that we know the targets, we only need to call `mvn` with the target we want. If you want to generate the very same Java archive file we use for releases, execute the command presented in Code 16.

Code 16: Building arara with Maven, first attempt.

```
$ mvn assembly:assembly
```

Actually, the command presented in Code 16, as the project structure is at the moment, will fail! Let me explain why: the application is not yet linked with the localized messages, so we need to convert our translation files into a correct format and then run the target in Maven. The error message after running `mvn assembly:assembly` presented in Code 17 gives us a hint on what we should do.

Code 17: The Maven error message about missing localization files.

```
Failed tests: testLocalizationFile(com.github.arara.AraraTest):
arara requires at least the default localization file
Messages.properties located at the translations/ directory in
the project repository. Rename Messages.input to
Messages.properties and copy the new file to the src/
directory, under com/github/arara/localization, and build
arara again.

Tests run: 1, Failures: 1, Errors: 0, Skipped: 0

[INFO] ------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------
```

Let's go into `arara/translations` and run the commands presented in Code 18. Since we are dealing with languages that require an encoding in UTF-8 while the localization files are set in ASCII, we need to run a conversion program in order to generate valid `.properties` files.

Now we can simply rerun the command presented in Code 16. Hopefully, we won't have trouble this time. Relax while Maven takes care of the building process. It might take a while, since all dependencies will be downloaded to your Maven repository. After a while, Maven will tell us that the project was built successfully!

After a successful build via Maven, we can now get the generated executable Java archive file `arara-3.0-with-dependencies.jar` which is inside the `arara/arara/target` directory, rename it to `arara.jar` and use it as we have seen in the previous chapters.

Code 18: Converting the localization files.

```
$ native2ascii -encoding utf8 Messages.input ../application/src
    /main/resources/com/github/arara/localization/Messages.
    properties
$ native2ascii -encoding utf8 Messages_de.input ../application/
    src/main/resources/com/github/arara/localization/Messages_de
    .properties
$ native2ascii -encoding utf8 Messages_es.input ../application/
    src/main/resources/com/github/arara/localization/Messages_es
    .properties
$ native2ascii -encoding utf8 Messages_fr.input ../application/
    src/main/resources/com/github/arara/localization/Messages_fr
    .properties
$ native2ascii -encoding utf8 Messages_it.input ../application/
    src/main/resources/com/github/arara/localization/Messages_it
    .properties
$ native2ascii -encoding utf8 Messages_pt_BR.input ../
    application/src/main/resources/com/github/arara/localization
    /Messages_pt_BR.properties
$ native2ascii -encoding utf8 Messages_ru.input ../application/
    src/main/resources/com/github/arara/localization/Messages_ru
    .properties
$ native2ascii -encoding utf8 Messages_tr.input ../application/
    src/main/resources/com/github/arara/localization/Messages_tr
    .properties
```

3.3 Notes on the installer and wrapper

The project directory has additional subdirectories regarding the arara cross-platform installer and the `.exe` wrapper. It's important to observe that only the build files are available, which means that you need to review the compilation process and make adjustments according to your directory structure.

The cross-platform installer Java archive file is generated with IzPack [4], while the `.exe` wrapper is built with Launch4J [3]. Both build files are written in plain XML, so you can easily adapt them to your needs. Sadly, the main purpose of this chapter is to cover the build process of arara itself and not its helper tools; if you want to generate your own wrapper or installer, please refer to the available documentation on how to build each

file. The build process is also very straightforward.

References

[1] *Apache Maven*. Software project management and comprehension tool. URL: http://maven.apache.org/ (cit. on p. 27).

[2] *Java Development Kit*. URL: http://www.oracle.com/technetwork/java/javase/downloads/index.html (cit. on p. 27).

[3] Grzegorz Kowal. *Launch4J, a cross-platform Java executable wrapper*. 2005. URL: http://launch4j.sourceforge.net/ (cit. on p. 30).

[4] Julien Ponge. *IzPack*. The project is developed by a community of benevolent contributors. 2001. URL: http://izpack.org/ (cit. on p. 30).

Chapter 4

IDE integration

The answer to "can Emacs..." is always "yes".

David Carlisle

This chapter covers the integration of arara with several integrated development environments. For obvious reasons, it's almost impossible for us to cover the full range of editors available nowadays, so we tried to focus only on a couple of them. If you use arara with an IDE other than the ones listed here, please let us know! It would be great to include your contribution in this user manual.

4.1 TeXworks

arara can be easily integrated with TeXworks [1], an environment for authoring TeX documents shipped with both TeX Live and MiKTeX. In this section, we will learn how to integrate arara and this cross-platform TeX front-end program.

First of all, make sure arara is properly installed in your operating system. Thankfully, it's very easy to add a new tool in TeXworks, just open the program and click in *Edit* → *Preferences...* to open the preferences screen (Figure 4.1).

The next screen is the TeXworks preferences (Figure 4.2). There are several tabs available. Navigate to the *Typesetting* tab, which contains two lists: the paths for TeX and related programs, and the processing tools. In the second list – the processing tools – click the *Plus (+)* button to add another tool.

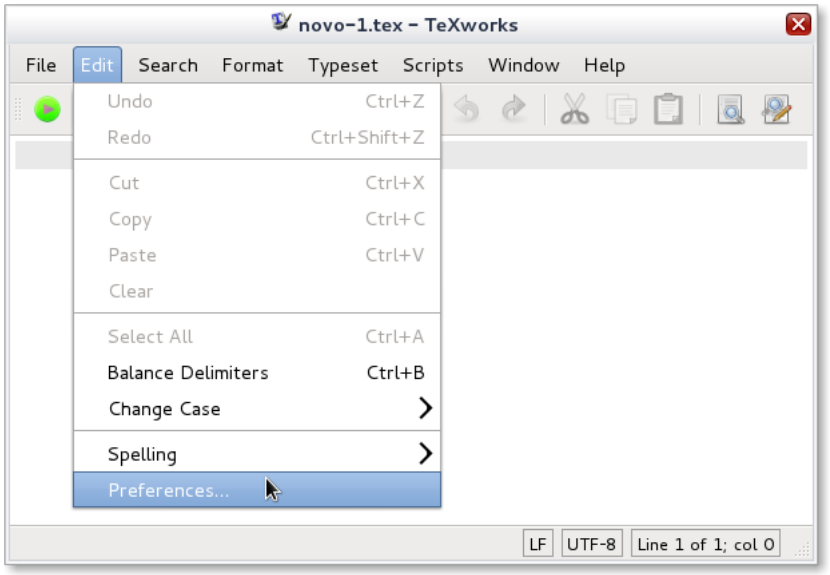

Figure 4.1: Opening the preferences screen in TEXworks.

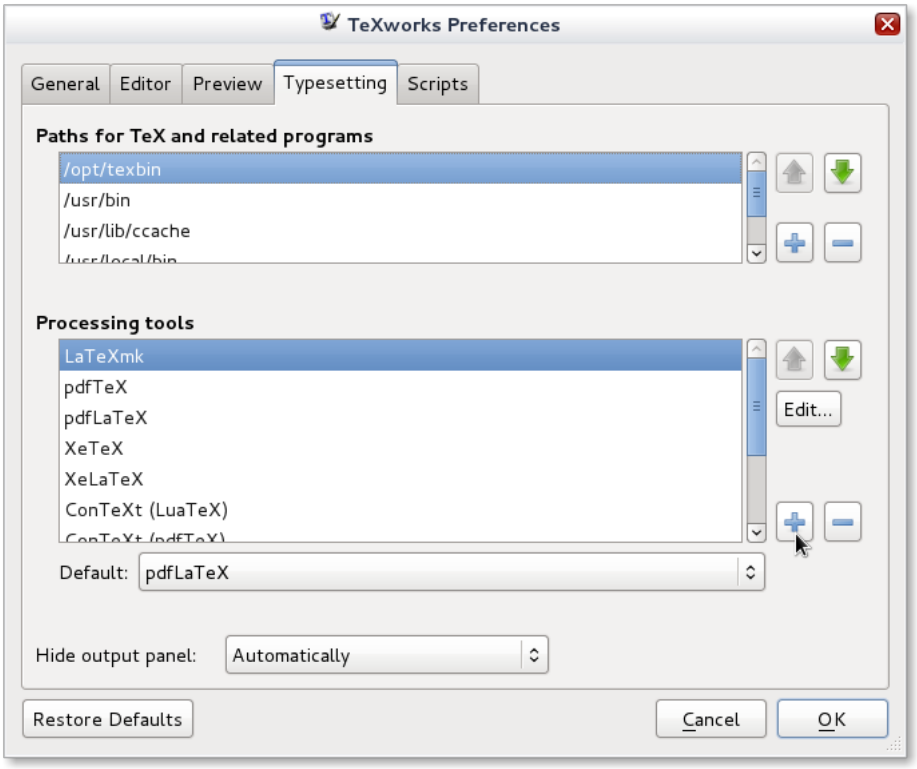

Figure 4.2: The TEXworks preferences screen.

We are now in the new tool screen (Figure 4.3). TEXworks provides an very straightforward interface for adding new tools; we just need to provide the tool name, the executable path, and the parameters. Table 4.1 helps us on what to type in each field. When done, just click *OK* and our new tool will be available.

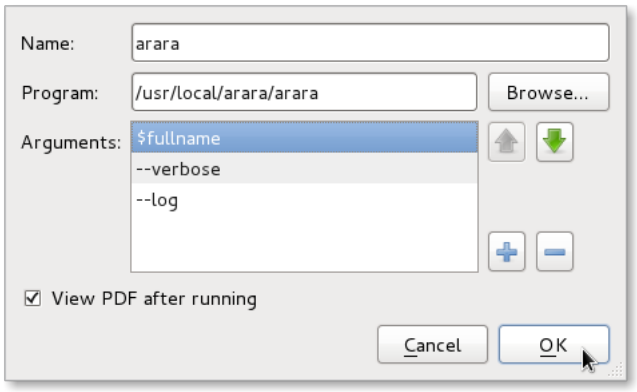

Figure 4.3: The new tool screen.

Field name	Value	Description
Name	`arara`	The tool name. You can actually type whatever name your heart desires. This value will be displayed in the compilation profile.
Program	`$ARARA_HOME/arara`	The full executable path. Just browse the filesystem and select the correct arara path. Observe that symbolic links are resolved to their full targets. For Windows, select the `.exe` wrapper; for Unix, select the bash script.

Arguments	`$fullname`
	`--verbose`
	`--log`
		The tool arguments. Note that you need to type one argument at a time, by clicking the *Plus (+)* button. The first argument is a TEXworks variable which will expand to the current filename. The second and third arguments are arara flags, discussed later, in Chapter 7.

Table 4.1: Configuring arara in TEXworks.

We are now back to the preferences screen (Figure 4.2). Hopefully, arara is in the list of processing tools. Just click *OK* to confirm the new

addition. Congratulations, now arara is available as a compilation profile in TEXworks (Figure 4.4).

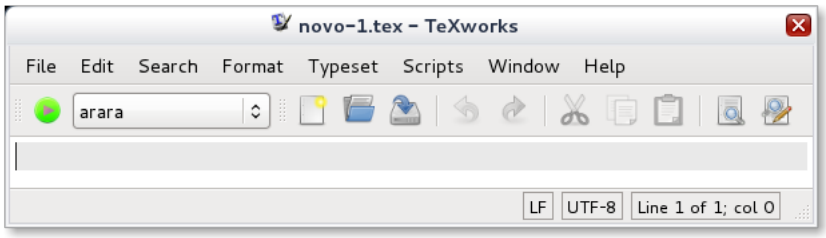

Figure 4.4: Using arara in the TEXworks compilation profile.

And we are done, arara is now integrated with TEXworks! Just make sure to select the correct profile when running the compilation process.

4.2 WinEdt

The following procedure is kindly provided by Harish Kumar. It's very easy to integrate arara with WinEdt, let's take a look at the steps.

Getting images for icons and toolbar

WinEdt uses 16 × 16 `.png` images for menu items, the toolbar and the tree control. They are available in `%B\Bitmaps\Images`[1] folder and they are automatically loaded on startup or later through the *Options Interface* (or `ReloadImages` macro function). Restricted users can place additional images in their `%b\Bitmaps\Images`[2] folder. At the moment all images have a 16 × 16 dimension and use 32-bit transparent `.png` format.

The images for the arara toolbar can be downloaded from the downloads area of the project repository. It would suffice to have a 16 × 16 `.png` image for WinEdt v7 while for WinEdt v6, one has to use 16 × 16 `.bmp` image. The downloaded images must be copied to `%B\Bitmaps\Images` or `%b\Bitmaps\Images` (depending upon the admin privileges). Once copied, WinEdt has to be restarted to load the images. Now the images should be available for use.

[1] `%B` maps to `C:\Program Files\WinEdt Team\WinEdt 7` for default installation.

[2] `%b` maps to `C:\Users\<username>\AppData\Roaming\WinEdt Team\WinEdt 7` for default installation

Adding a menu entry

The following steps describe how to add a menu entry for arara in WinEdt v6 and v7.

1. Go to *Options* → *Options Interface*. A side window will appear on the left side as shown in Figure 4.5.
2. From the *Menus and Toolbar* drop down list, select *Main Menu* and double click to open the file `Main Menu.ini`.
3. In the `Main Menu.ini` file, type the code presented in Code 19 somewhere in the file.
4. Save the file. Now the current script has to be loaded by clicking the *Load current script* button, which is the first button in the tool bar in *Options Interface* window shown in Figure 4.5.
5. Now in the WinEdt *TEX* menu, a submenu called *Arara* should be visible and functional (Figure 4.6).
6. From the *Menus and Toolbar* drop down list, select *Toolbar* and double click to open the file `Toolbar.ini`.
7. In the `Toolbar.ini` file, type the following line, somewhere in the file: `BUTTON="arara"`.
8. Save the file `Toolbar.ini`. Now the current script has to be loaded by clicking the *Load current script* button, which is the first button in the tool bar in *Options Interface* window (Figure 4.5).
9. Now a button for arara should be visible as shown in Figure 4.7.

Code 19: Adding an entry to arara in `Main Menu.ini`.

```
ITEM="Arara"
CAPTION="Arara"
IMAGE="arara16"
SAVE_INPUT=1
MACRO=:RunConsole('arara "\%F"','\%P','arara...');
REQ_FILTER=:"\%!M=TeX"|"\%!M=TeX:STY"|"\%!M=TeX:AUX"
```

And we are done! arara is successfully integrated in WinEdt. Now you can add directives to your files and click the buttons to trigger the execution.

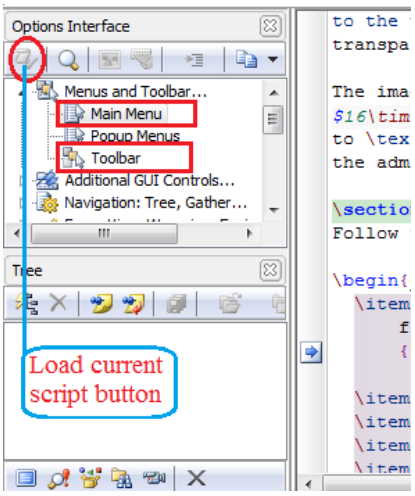

Figure 4.5: The *Options Interface* window in WinEdt.

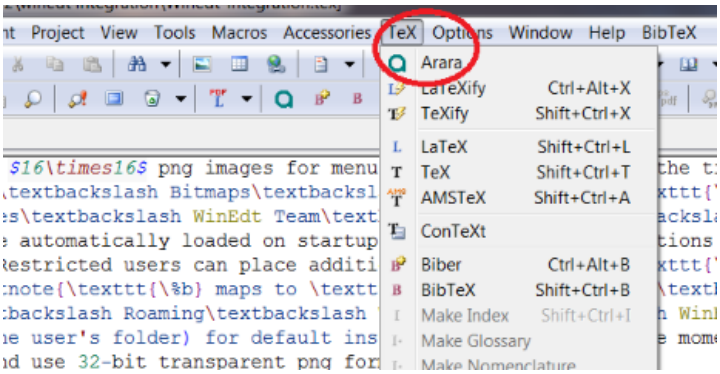

Figure 4.6: The WinEdt *TEX* menu.

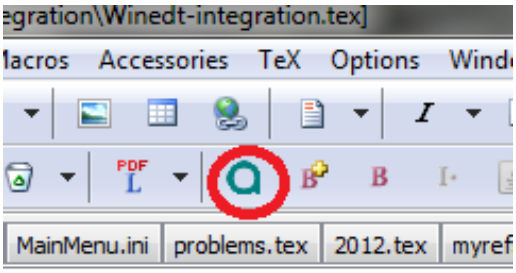

Figure 4.7: The arara button in WinEdt.

4.3 Inlage

The following procedure is kindly provided by Harish Kumar. It's very easy to integrate arara with Inlage, let's take a look at the steps.

Inlage v4

The following steps describe how to add a menu entry for arara in Inlage v4. It's an easy procedure.

1. Go to *Build* → *User Commands*. A window named *Edit Commands* will appear as shown in Figure 4.8.
2. Now press the *Plus (+)* button to get the *Add Command* window as shown in Figure 4.9.
3. Under the *Name* textfield, type `arara` and under the *Command Line* textarea, type `arara \%f`. Now the new configuration should be saved using the *Save* button. Now an entry for arara should be visible as seen in Figure 4.10.
4. These settings must be then updated using the *Update* button in the *Edit Commands* window shown in Figure 4.8.
5. Now a menu entry for arara should be visible in *Build* → *Execute* → *arara*.

That's it, arara is now successfully integrated in Inlage v4. Have fun!

Inlage v5

The following steps describe how to add a menu entry for arara in Inlage v5. It's an easy procedure.

1. Go to *Build* → *Compiler Options...*. A *Settings* window will appear as shown in Figure 4.11.
2. Under *Profiles*, create a new profile clicking the *New* button. A new window will open; now let's type the name for the new profile as `arara`, as shown in Figure 4.12. Now press *Okay*.
3. Under *Compiler Order*, press *Add Compiler*, as shown in Figure 4.13.
4. Under *Binaries*, change the following:

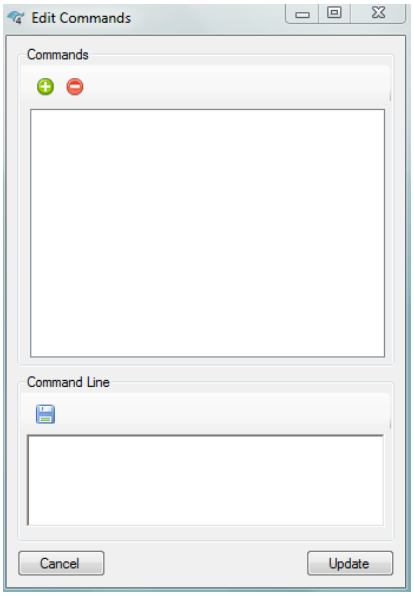

Figure 4.8: The *Edit Commands* window in Inlage.

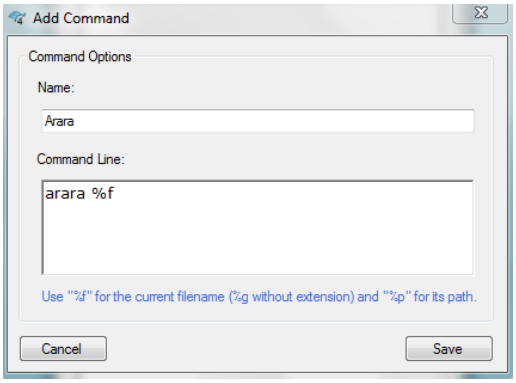

Figure 4.9: The *Add Command* window in Inlage.

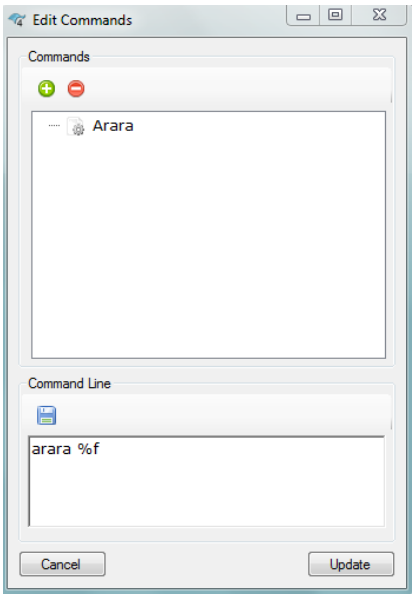

Figure 4.10: arara added to the *Edit Commands* window in Inlage.

a) In *Implementation*, choose *Custom*.
b) In *Binary Name*, choose the executable for arara using the *Browse* button.
c) In *Parameters*, you can choose the parameters, such as `--verbose`.
d) In *Target File*, choose *Active File/Masterfile*.

5. Save these settings. After the previous steps, you should get a menu for arara in Inlage v5 as seen in Figure 4.14.

That's it, arara is successfully integrated with Inlage v5. Have fun!

4.4 TEXShop

Integrating arara with TEXshop is probably one of the easiest procedures. Simply open your terminal, go to `~/Library/TeXShop/Engines` and create a file named `arara.engine` with the content presented in Code 20.

Now, we need to add execute permissions to our newly created file. A simple `chmod +x arara.engine` will do the trick. Now, open TEXshop and you will see arara available for use in the list of engines (Figure 4.15).

No more steps are needed, arara is successfully deployed in TEXshop. If you want to remove arara from TEXshop, just remove `arara.engine` from the `Engines` directory.

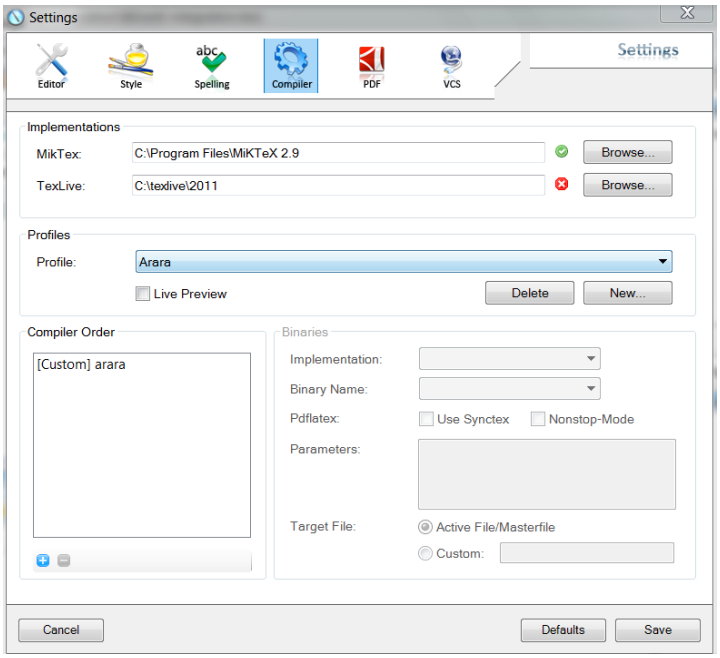

Figure 4.11: The *Settings* window in Inlage.

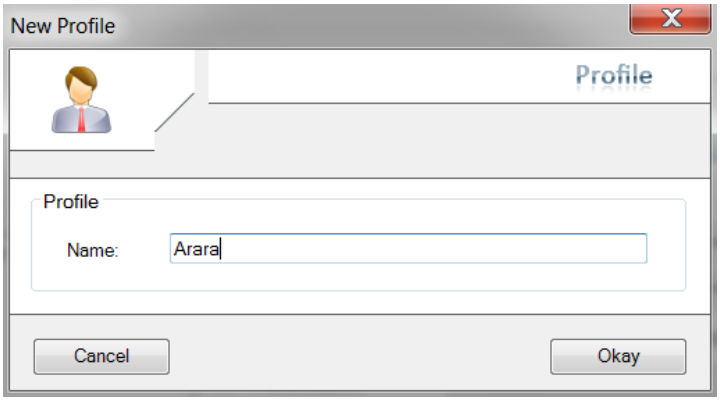

Figure 4.12: Adding a new profile in Inlage.

Code 20: `arara.engine`

```
#!/bin/bash
export PATH=/usr/texbin:/usr/local/bin:${PATH}
arara "$1"
```

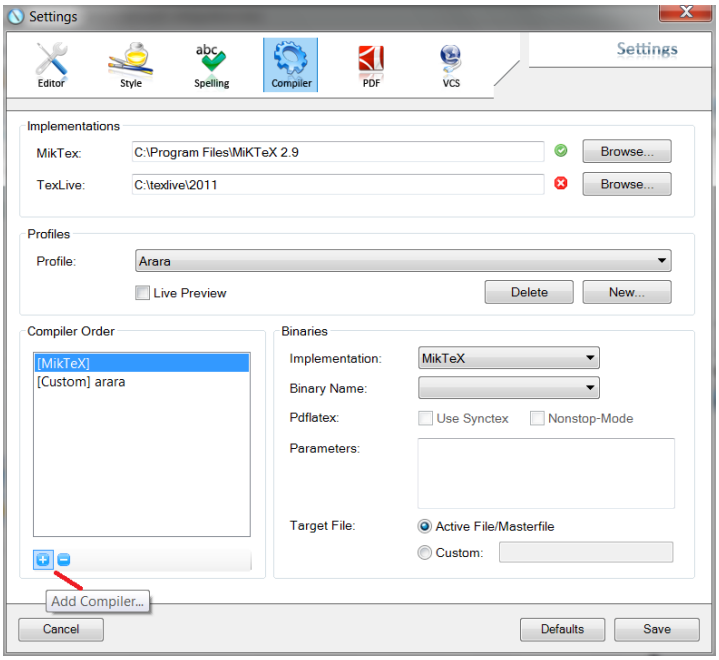

Figure 4.13: Adding a compiler in Inlage.

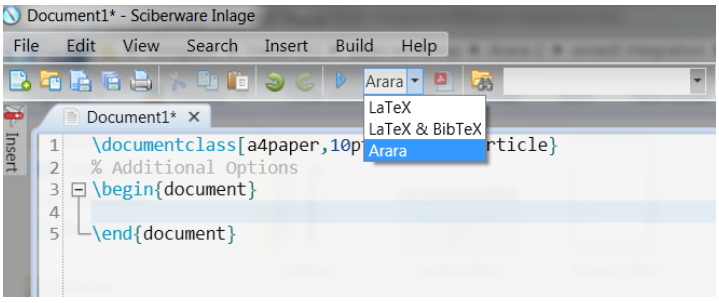

Figure 4.14: `arara` added to the menu in Inlage.

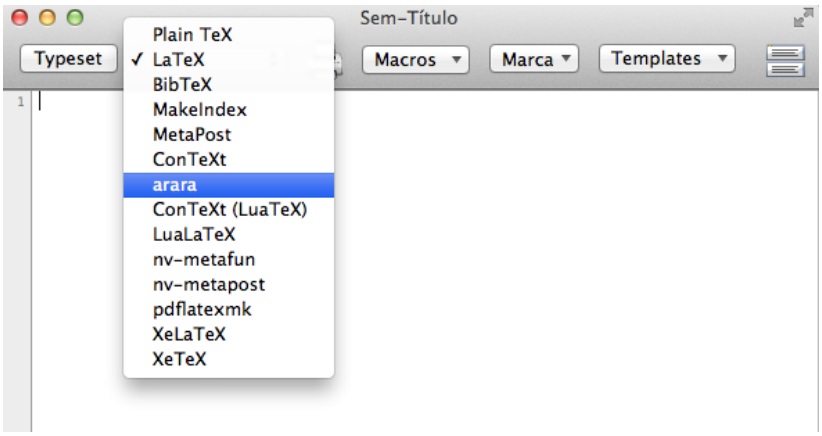

Figure 4.15: arara available in TEXshop.

4.5 TEXnic Center

TEXnic Center has also an easy integration with arara. The first step is to go to *Build* → *Define Output Profiles...*. The *Profiles* window should open, as shown in Figure 4.16.

Let's now create a new profile! Click the *Add* button. A window will pop up and ask for the new profile name (Figure 4.17). Type `arara` as the profile name and click *OK*.

The last step is the easiest one. With the `arara` profile selected, check the box which says *Run (La)TEXin this profile*, then enter the full path to `arara` in the textbox named *Path to the (La)TEXcompiler* and write `--log --verbose "%wm"` in the texbox named *Command line arguments to pass to the compiler*; click *OK*, as shown in Figure 4.18.

That's it, now arara is integrated with TEXnic Center! Just make sure to select `arara` in the dropdown list of available profiles.

References

[1] Jonathan Kew, Stefan Löffler, and Charlie Sharpsteen. *TEXworks: lowering the entry barrier to the TEX world*. 2009. URL: http://www.tug.org/texworks/ (cit. on p. 33).

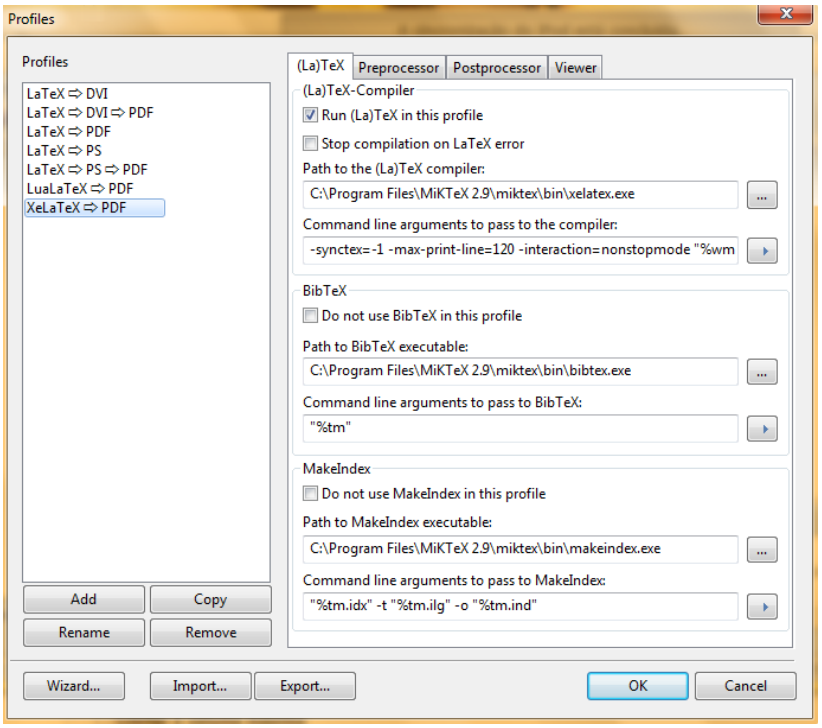

Figure 4.16: The *Profiles* window in TEXnic Center.

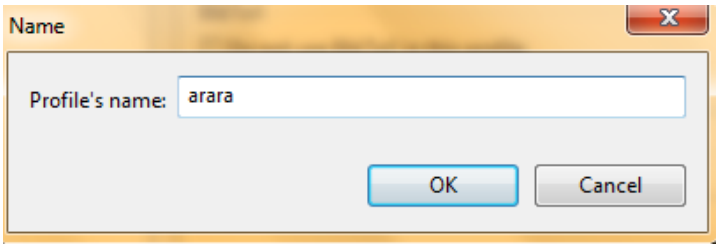

Figure 4.17: Creating a new profile in TEXnic Center.

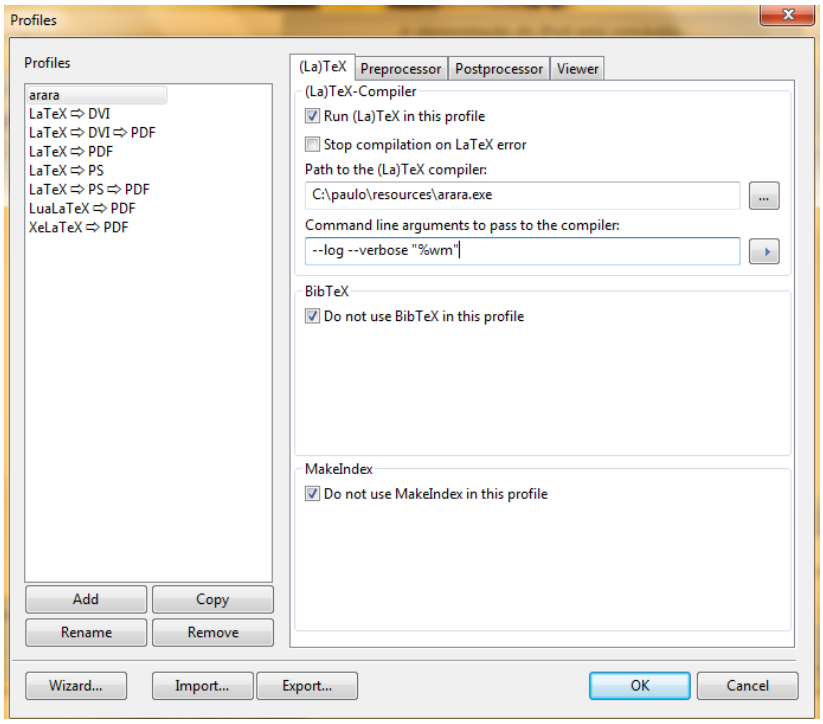

Figure 4.18: Configuring arara in TEXnic Center.

Chapter 5

Important concepts

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth

Time for our first contact with arara! It's important to understand a few concepts in which arara relies before we proceed to the usage itself. Do not worry, these concepts are easy to follow, yet they are vital to the comprehension of the application and the logic behind it.

5.1 Rules

Do you remember `mydoc.tex` from Code 1 in page 4? 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 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 concept to be introduced in Section 5.2.

A rule is a plain text file written in the YAML format [2]. I opted for this format because it's cleaner and more intuitive to use than other markup languages, besides of course being a data serialization standard for all programming languages. As a bonus, the acronym *YAML* rhymes with the word *camel*, so arara is heavily environmentally friendly[1].

[1] Perl, I'm looking at you.

The default rules, that is, the rules shipped with arara, are placed inside a special subdirectory named `rules/` inside `ARARA_HOME`. We will learn in Section 6.1 that we can add an arbitrary number of paths for storing our own rules, in order of priority, so don't worry with the location of the default rules, although it's important to understand and acknowledge their existance. The basic structure of an arara rule is presented in Code 21.

Code 21: `makefoo.yaml`, a basic structure of an arara rule.

```
1 !config
2 # I am a comment
3 identifier: makefoo
4 name: MakeFoo
5 command: makefoo @{file}
6 arguments: []
```

The `!config` keyword (line 1) is mandatory and it must be the first line of any arara rule. Note that the format also accepts comments (line 2) by simply starting a line with the `#` symbol. The following keys are defined:

`identifier`
: This key (line 3) 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 4) 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` in the output when dealing with this task.

`command`
: This key (line 5) contains the system command to be executed. You can use virtually any type of command, interactive or noninteractive. But beware: if arara is running in silent mode, which is the default behaviour, an interactive command wich might require the user input will be halted and the execution will fail. Don't despair, you can use a special `--verbose` flag with arara in order to interact with such commands – we will talk about flags in Chapter 7. There are cases in which you might want to have a list of commands instead of a single one; arara has support for multiple commands inside one rule, we just

need to replace `command` by `commands` and provide a list of commands to be executed, as seen in Code 22. You probably noticed a strange element `@{file}` in the `command` line: this element is called *orb tag*. For now, just admit these elements exist. We will come back to them later on, in Section 5.3, I promise.

`arguments`
: The `arguments` key (line 6) denotes a list of arguments for the rule command. In our example, we have an empty list, denoted as `[]`. You can define as many arguments as your command requires. Please check Code 23 for an example of a list of arguments.

There are cases in which we need to run more than just one command for a certain rule. Take, for example, the `frontespizio` rule released with arara: when using the `frontespizio` package[2], the document has to processed by the choosen engine, say `pdflatex`, no less than three times; if `latex` is used, there's an additional run of `dvips`. In that case, the logic is enclosed inside the rule, so there's no need to write every compilation step as required by the package as directives in the source code; a simple call to `frontespizio` is enough to generate the proper results.

If you need to run more than one command inside a rule, replace the `command` identifier by `commands` and add one command per line, preceeded by `-` to indicate an item in the list. Code 22 presents a sample `makefoobar` rule which runs the `makefoo` program two times, followed by one run of the `makebar` program.

Code 22: `makefoobar.yaml`, an arara rule with multiple commands.

```
1 !config
2 identifier: makefoobar
3 name: MakeFooBar
4 commands:
5 - makefoo @{file}
6 - makefoo @{file}
7 - makebar @{file}
8 arguments: []
```

For more complex rules, we might want to use arguments. Code 23 presents a new rule which makes use of them instead of an empty list as we saw in Code 21.

[2] http://ctan.org/pkg/frontespizio, written by Enrico Gregorio.

Code 23: `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 @{parameters.one}
8 - identifier: two
9   flag: -j @{parameters.two}
```

For every argument in the `arguments` 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. Note that we have other orb tags in the arguments definitions, `@{parameters.one}` and `@{parameters.two}`; we will discuss them later on, in Section 5.3. Just to give some context, `parameters` is a special keyword which maps the elements available in the directives. For example, if we have `one: 1` in a directive, `parameters.one` will resolve to `1`. The argument `flag` value is only triggered, that is, resolved, if and only if there's an explicit directive argument. Say, if `one` is not defined as a directive argument, the `flag` value of the argument `one` will be resolved to an empty string. There's a way of overriding the default empty string value when a directive argument is not specified, which is done by using the `default` key. By the way, the `flag` key is not really mandatory, but for most of the rules, you'll need it. At least one of the `flag` and `default` keys is mandatory.

If we need to set a default value other than an empty string to a rule argument, we can use the `default` key. When a rule argument just needs a default value, you can safely ignore the `flag` key and rely on the `default` key. If you need to map a directive argument into a rule argument without

falling back to a default value different than an empty string, just use the `flag` key. Now, if you need mapping and fallback, stick with both keys.

For now, we need to 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, as we mentioned before already: directives are mapped to rules through orb tags. Don't worry, I'll explain how things work.

5.2 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 file and extract the directives. A directive should be placed in a line of its own, in the form `% arara: <directive>` – actually, we will see in Section 6.3 that the prefix search can be altered. There are two types of directives:

empty directive
: An empty directive, as the name indicates, has only the rule identifier, as we seen in Section 5.1. Lines 1 and 3 of Code 24 show an example of empty directives. Note that you can suppress arguments (line 3 in contrast to line 2), but we will see that arara assumes that you know exactly what you are doing. The syntax for an empty directive is `% arara: makefoo`.

parametrized directive
: A parametrized directive has the rule identifier followed by its arguments. Line 2 of Code 24 shows an example of a parametrized directive. It's very important to mention that the arguments are mapped by their identifiers and not by their positions. The syntax for a parametrized directive is `% arara: makefoo: { arglist }`. The argument is in the form `arg: value`; a list of arguments and their respective values is separated by comma.

The arguments are defined according to the rule mapped by the directive. For example, the rule `makebar` (Code 23) 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

Code 24: Example of directives in a .tex file.

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

will be ignored. arara always look for a line that, after removing the leading and trailing spaces, starts with a comment `%` and has the keyword `arara:` in it. In Section 6.3, we will learn how to override this search pattern, but the `arara:` keyword is always immutable.

Directives are mapped to rules. In Section 5.3 we will learn about orb tags and then revisit rules and directives. I hope the concepts will be clearer since we will understand what an orb tag is and how it works. How about a nice cup of coffee?

5.3 Orb tags

When I was planning the mapping scheme, I opted for a templating mechanism. I was looking for flexibility, and the powerful MVEL expression language [1] was perfect for the job. I could extend my mapping plans by using 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 the `name` variable resolving to `Paulo` will be expanded to the string `Hello, my name is Paulo`. Cool, isn't it? Code 25 presents a few examples on how orb tags are expanded.

In the first example of Code 25, `@{name}` simply indicates the expansion of the variable into its value, so the output is a concatenation of the text with the variable value. The second example is a conditional test, that is, whether the `name` variable has its value equals to `Paulo`; the result of this evaluation is then expanded, which is `true`. The third example presents a more complex construction: since `name` holds a string, MVEL resolves this variable to a `String` object and automatically all methods from the `String` class in Java are available to the variable, so the method `toUpperCase()` is called in order to make all characters in the string to be capitalized, and the output is presented. The fourth and last example presents a ternary oper-

Code 25: A few examples on how orb tags are expanded.

```
# always consider: name = Paulo

In[1]: Hello, my name is @{name}.
Out[1]: Hello, my name is Paulo.

In[2]: @{name == "Paulo"}
Out[2]: true

In[3]: @{name.toUpperCase()}
Out[3]: PAULO

In[4]: Hello, I am @{name == "Paulo" ? "John" : "Mary"}.
Out[4]: Hello, I am John.
```

ation, which starts with a conditional to be evaluated; if this test evaluates to true, the first string is printed, with the second string being printed in case the test is false.

When mapping rules, every command argument will be mapped to the form `@{identifier}` with value equals to the content of the `flag` key. The `@{identifier}` orb tag might hold the value of the `default` key instead, if the key is defined and there were no directive parameters referring to `identifier`. There are three reserved orb tags, `@{file}`, `@{item}` and `@{parameters}` – actually, that's not true, there's a fourth reserved orb tag which plays a very special role in arara – `@{SystemUtils}` – but we will talk about it later on. The `@{file}` orb tag refers to the filename argument passed to arara. The `@{file}` value can be overriden, but we will discuss it later. The second reserved orb tag `@{item}` refers to a list of items, in case the rule might use some sort of list iteration, discussed later on. The third reserved orb tag `@{parameters}` is a map which can expand 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 @{parameters.one}` 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`, unless of course the current argument doesn't have a `default` elements in its definition. The whole procedure is summarized as follows:

1. arara processes a file named `mydoc.tex`.
2. A directive `makebar: { one: hello }` is found, so arara will look up the rule `makebar.yaml` (Code 23) inside the default rules directory.
3. The argument `one` is defined and has value `hello`, so the corresponding `flag` key will have the orb tag `@{parameters.one}` 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, since there's no `default` key in the argument definition.
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 value of the `@{file}` orb tag, use the `files` key, like `makebar: { files: [ thedoc.tex ] }`. This will result in `makebar thedoc.tex` instead of `makebar mydoc.tex`. The very same concept applies to the other reserved directive key named `items`, which is also a list, and the expansion happens in the `@{item}` orb tag.

If you provide more than one element 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`. If you happen to have a rule which makes use of both `files` and `items` in the directive, you'll end up with a cartesian product of those two lists.

References

[1] Mike Brock. *MVEL, the MVFLEX Expression Language*. MVEL is a powerful expression language for Java-based applications. URL: http://mvel.codehaus.org/ (cit. on p. 52).

[2] *YAML*. 2001. URL: http://www.yaml.org/ (cit. on p. 47).

Chapter 6

Configuration file

An algorithm must be seen to be believed.

Donald Knuth

arara has support for an optional configuration file in order to enhance and override some settings of the application without the need of delving into the source code. The optional configuration file has to reside inside the user home directory, which is usually `C:\Users\Username` for Windows Vista and superior, or `~/username` for the Unix world, under the name `araraconfig.yaml`. arara always looks for a configuration file during every execution. In fact, `araraconfig.yaml` is just a plain text file written in the YAML format, starting with the `!config` line and at with least one of the three settings presented in the following sections. The order doesn't matter, as long as they are consistent.

6.1 Search paths

When looking for rules, arara always searches the default rule path located at `ARARA_HOME/rules`; if no rule is found, the execution halts with an error. It's not wise to mess with the default rule path, so we use the configuration file to add search paths, that is, a list of directories in which arara should look for rules. An example of a new search path is presented in Code 26.

According to Code 26, from now on, arara will look for rules first in the `/home/paulo/rules`; if the rule is not found, then the search falls back to the default search path located at `ARARA_HOME/rules`. We can even add an arbitrary number of paths, as seen in Code 27.

Code 26: An example of a new search path for the configuration file.

```
1 !config
2 paths:
3 - /home/paulo/rules
```

Code 27: An arbitrary number of paths added in the configuration file.

```
1 !config
2 paths:
3 - /home/paulo/rules
4 - /opt/arara/rules
5 - /home/paulo/myrules
```

The items order defines the search priority. arara also features a special orb tag for search paths named `@{userhome}` which maps the variable to the user home directory, for example, `/home/paulo`, according to your operating system. But before we proceed, a word on the YAML format.

Sadly, we can't start values with `@` because this symbol is reserved for future use in the YAML format. For example, `foo: @bar` is an invalid YAML format, so the correct usage is to enclose it in quotes: `foo: '@bar'` or `foo: "@bar"`. We also need to enclose our strings with quotes in arara, but now we can save them by simply adding the `<arara>` prefix to the value. In other words, `foo: <arara > @bar` is correctly parsed; when that keyword in that specific position is found, arara removes it. That means that the orb tag presented in Code 28 will be correctly parsed.

Code 28: Using the special orb tag for mapping the home directory in the configuration file.

```
1 !config
2 - '@{userhome}/rules'
3 - /opt/arara/rules
4 - <arara> @{userhome}/myrules
```

It's important to observe that the `<arara>` prefix is also valid in the rules context, presented in Section 5.1. The idea of using this prefix is to actually ease the writing of rules that involve quoting without the need of escaping all internal quotes or even alternating between single and double quotes. It's also a way of writing cleaner rules.

6.2 Language

arara currently features localized messages in English, French, Italian, German, Spanish, Brazilian Portuguese, Russian and Turkish. The default language fallback is English, but we can easily change the language by adding `language: <code>` to the configuration file, as seen in Code 29. The list of languages and codes is presented in Table 6.1.

Code 29: Changing the language in the configuration file.

```
1 !config
2 language: en
```

Table 6.1: Languages and codes.

Language	Code
English	`en`
Brazilian Portuguese	`ptbr`
Italian	`it`
Spanish	`es`
German	`de`
French	`fr`
Russian	`ru`
Turkish	`tr`

There's also a `--language` command line flag which has a higher priority, so it overrides the configuration file setting, if any. Beware of the terminal you use; the Windows command prompt has serious troubles in understanding UTF-8. You probably won't run into problems with the applications shipped in Mac or Linux.

6.3 File patterns

arara accepts the following filetypes: `tex`, `dtx` and `ltx`. If no file extension is provided in the command line, for example, calling `arara mydoc` instead

of `arara mydoc.tex`, the application will automatically look for files that match the filetypes in that specific order, that is, `mydoc.tex`, `mydoc.dtx` and `mydoc.ltx`. Let's say we want to change the order by promoting `dtx` to the first match; we can easily achieve that by rearranging the items of the list of filetypes in the configuration file according to Code 30.

Code 30: Rearranging the list of filetypes in the configuration file.

```
1 !config
2 filetypes:
3 - extension: dtx
4 - extension: tex
5 - extension: ltx
```

The `filetypes` key in the configuration file is actually way more powerful than the example shown in Code 30. Before we continue, let's start with some basics. Consider the three directives presented in Code 31.

Code 31: Three directives with different formatting patterns.

```
1 % arara: foo
2    % arara: foo
3 %     arara: foo
4 \documentclass{book}
5 ...
```

The default setting for arara is to recognize the three directives shown in Code 31. In other words, the search pattern for all the three extensions is `^(\\s)*%\\s+` plus `arara:\\s` which is immutable, of course. Let's say that, for the `dtx` format, you want arara to look for directives that have no spaces in the beginning of the line, that is, the line must start with only one percentage sign followed by at least one space and the default prefix. We can easily achieve such requirement by adding a `pattern` element to our list, as presented in Code 32.

Now, only the first directive of Code 31 is recognized, if the analyzed file has the `.dtx` extension. All other extensions – `.tex` and `.ltx` – will follow the default search pattern.

We can also extend arara to analyze files with arbitrary extensions. As an example, let's suppose we have a sample `hello.c` file, presented in Code 33. Note that the code was omitted for obvious reasons, since we are interested in the header.

Code 32: Changing the search pattern for `.dtx` files.

```
!config
!config filetypes:
- extension: dtx
  pattern: ^%\\s+
- extension: tex
- extension: ltx
```

Code 33: A sample `hello.c` code.

```
// arara: gcc
#include <stdio.h>
...
```

We can add the `.c` extension to be recognized by arara by simply adding the extension and search pattern entries in the configuration file, as presented in Code 34.

Code 34: Adding support for `.c` files in the configuration file.

```
!config
filetypes:
- extension: c
  pattern: ^\\s*//\\s*
```

Done, now arara can support `.c` files! We can run `arara hello.c` and have our code compiled, provided we have a `gcc` rule, of course. The extensions list will be `.tex`, `.dtx`, `.ltx` and `.c`. If you want to change the order, it's a matter of rearranging the items, as shown in Code 35.

From now on, the `.c` has priority over all other extensions. It's very important to note that for customized extensions, the `pattern` key is mandatory. For default extensions, use the `pattern` key if and only if you want to override the search pattern.

arara comes with a rule for Sketch [1], written by Sergey Ulyanov. We can easily add `% arara: sketch: { files: [ drawing.sk ] }` and Sketch will be properly called. Let's say we want to make arara recognize Sketch files; it's just a matter of adding the extension and the search pattern in our configuration file, as presented in Code 36.

Code 35: Rearranging items of arbitrary extensions in the configuration file.

```
1 !config
2 filetypes:
3 - extension: c
4   pattern: ^\\s*//\\s*
5 - extension: tex
6 - extension: dtx
7 - extension: ltx
```

Code 36: Adding support for Sketch files in the configuration file.

```
1 !config
2 filetypes:
3 - extension: sk
4   pattern: ^(\\s)*[%#]\\s+
```

Now arara supports `.sk` files! We can write a sample Sketch file (borrowed from the documentation) presented in Code 37 and add a `sketch` directive. The comments in the Sketch language allow both `%` and `#` symbols at the beginning of the line.

Code 37: `drawing.sk`, a sample Sketch file.

```
1 % arara: sketch
2 polygon(0,0,1)(1,0,0)(0,1,0)
3 line(-1,-1,-1)(2,2,2)
```

With the new settings presented in Code 36, we can run `arara drawing` or `arara drawing.sk` (Code 37) and Sketch will be properly executed through arara with no problems.

References

[1] Eugene Ressler. *Sketch*. Sketch is a small, simple system for producing line drawings of two or three-dimensional objects and scenes. URL: http://www.frontiernet.net/~eugene.ressler (cit. on p. 59).

Chapter 7

Running arara

Never trust a computer you can't throw out a window.

Steve Wozniak

Now that we have learned some basics, it's time to run arara! Thankfully, the application is very user-friendly; if something goes wrong, we can easily find out what happened through messages and the log file.

7.1 Command line

arara has a very simple command line interface. A simple `arara mydoc` does the trick – provided that `mydoc` has the proper directives. The default behaviour is to run in silent mode, that is, only the name and the execution status of the current task are displayed. The idea of the silent mode is to provide a concise output. Sadly, in some cases, we want to follow the compilation workflow and even interact with a command which requires user input. If you have an interactive command, arara won't even bother about it: the execution will halt and the command will fail. Well, that's the silent mode. Thankfully, arara has a set of flags that can change the default behaviour or even enhance the compilation workflow. Table 7.1 shows the list of available arara flags, with both short and long options.

arara can recognize three types of files based on their extension, in this order: `.tex`, `.dtx` and `.ltx`. Other extensions are not recognized, unless of course you provide the correct mapping for them in the configuration file, as discussed in Section 6.3.

Flag		Behaviour
`-h`	`--help`	This flag prints the help message, as seen in Code 11, and exits the application. If you run `arara` without any flags or a file to process, this is the default behaviour.
`-L c`	`--language c`	The `--language` flag sets the language of the current execution of arara, where `c` is the language code presented in Table 6.1, on page 57. Note that this flag has higher priority than the language set in the configuration file.
`-l`	`--log`	The `--log` flag enables the logging feature of arara. All streams from all commands will be logged and, at the end of the execution, an `arara.log` file will be generated. The logging feature is discussed in Section 7.3.
`-t n`	`--timeout n`	This flag sets an execution timeout for every task. If the timeout is reached before the task ends, arara will kill it and interrupt the processing. The n value is expressed in milliseconds.
`-v`	`--verbose`	The `--verbose` flag enables all streams to be flushed to the terminal – exactly the opposite of the silent mode. This flag also allows user input if the current command requires so. The user input interaction is possible thanks to the amazing Apache Commons Exec library [2].
`-V`	`--version`	This flag, as the name indicates, prints the current arara version and exits the application.

Table 7.1: The list of available arara flags.

The combination of flags is very useful to enhance the TEX experience. They can provide nice features for integrating arara with TEX IDEs, as seen in Chapter 4. Note that both `--log` and `--verbose` flags are the most common combo to use in an IDE, so we can have both terminal and file output at the same time without any cost.

7.2 Messages

Messages are the first type of feedback provided by arara. They are basically related to rules, directives and configuration settings. Bad syntax, nonexisting rules, malformed directives, wrong expansion, arara tries to

tell you what went wrong. Those messages are usually associated with errors. We 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. For example, if you try to call a rule that executes a nonexisting `makefoo` command, arara will complain about it.

These messages usually cover the events that can happen during the preprocessing phase. Don't panic, arara will tell you what happened. Of course, an error halts the execution, so we need to fix the reported issue before proceeding. Note that arara can also complain about nonexisting commands – in this case, the error will be raised in runtime, since it's an underlying operating system dependency.

If you use the `--language` flag or set up the `language` key in the configuration file, arara will be able to display localized messages according to the provided language code. In other words, users will be able to read messages from the application in languages other English. Currently, arara is able to display messages in English, Brazilian Portuguese, Spanish, German, Italian, French, Russian and Turkish. Have fun!

7.3 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 [1] 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 both `stdout` and `stderr` is referred as *Output 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 in the same output stream. Read the log entries carefully. A excerpt of the resulting `arara.log` from `arara helloindex --log` is shown in Code 38 – 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.

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 the execution and fix the offending line in your code.

7.4 Command output

Even when the `--log` flag is enabled, arara still runs in silent mode. There's a drawback of this mode: if there's an interactive command wich requires the user input, arara will simply halt the task and the execution will fail. We need to make `stdin` – the standard input stream – available for us. Thanks to the amazing Apache Commons Exec library [2], arara can also provide an access layer to the standard input stream in order to interact with commands, when needed. We just need to use a special `--verbose` flag.

It's important to note that both `--log` and `--verbose` flags can be used together; arara will log everything, including the input stream. I usually recommend those two flags when integrating arara with TEX IDEs, like we did in Chapter 4.

Code 38: `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 helloindex.
    tex
09 Abr 2012 11:27:59.435 TRACE CommandTrigger - 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 - Output 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 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.
```

References

[1] *Standard streams*. Wikipedia, the free encyclopedia. URL: http://en.wikipedia.org/wiki/Standard_streams (cit. on p. 63).

[2] The Apache Software Foundation. *Apache Commons Exec*. 2010. URL: http://commons.apache.org/exec/ (cit. on pp. 62, 64).

Part II

For authors

Chapter 8

Quick start

Snakes! Why did it have to be snakes?

Indiana Jones, *Raiders of the Lost Ark* (1981)

This chapter covers a quick start of arara, including an overview of the predefined rules and some notes on how to properly organize directives in the source code.

8.1 Predefined rules

Let's take a look on the predefined rules and a brief description of their parameters. Note that these rules are constantly updated; the most recent versions are available in the project repository.

For convenience, we will use `yes` and `no` for representing boolean values. Note that you can also use other pairs: `on` and `off`, and `true` and `false`. These values are also case insensitive, so entries like `True` or `NO` are valid.

Note that the `latex`, `pdflatex`, `xelatex` and `lualatex` rules have a `shell` parameter resolving to `--shell-escape`. This flag is also available in MiKTeX, but as an alias to the special `--enable-write18` flag. If you want to use arara with an outdated MiKTeX distribution which doesn't support the `--shell-escape` alias, make sure to edit the predefined rules accordingly – these rules are located inside `$ARARA_HOME/rules` – and replace all occurrences of `--shell-escape` by `--enable-write18`. Another option is to add another search path in the configuration file with modified rules,

since custom search paths have higher priority than the default rules directory. If you use TEX Live or a recent MikTEX installation, there's no need to edit the rules, since the `--shell-escape` flag is already available.

biber

Description

This rule maps biber, calling the `biber` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: biber
```

Parameters

`options`

This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

BIBTEX

Description

This rule maps BIBTEX, calling the `bibtex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: bibtex
```

Parameters

`options`

This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

Clean

Description

This rule maps the removal command from the underlying operating system. There are no parameters for this rule, except the the reserved directive key `files` which *must* be used. If `files` is not used in the directive, arara will simply ignore this rule.

Syntax

```
% arara: clean
```

dvips

Description

This rule maps dvips, calling the `dvips` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: dvips
```

Parameters

`output`

This parameter is used to set the output PostScript filename. If not provided, the default output name is set to `@{getBasename(file)}.ps`.

`options`

This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

frontespizio

Description

This rule maps a compilation chain defined in `frontespizio`, a package written by Enrico Gregorio; it calls a defined TEX engine three times with the proper parameters, when available. All parameters are optional. If no engine is provided, `pdflatex` is used as default. When `latex` is the chosen engine, there's an additional call to dvips.

Syntax

```
% arara: frontespizio
```

Parameters

`engine`

This parameter is used to set the TEX engine. If not provided, `pdflatex` is used as a default value.

LaTeX

Description

This rule maps LaTeX, calling the `latex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: latex
```

Parameters

`action`

This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`

This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`

This parameter is defined as boolean and sets the generation of SyncTeX data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`draft`

This is a boolean parameter which sets the draft mode, that is, no PDF output is generated. When value set to true, the draft mode is enabled, while false disables it. If not defined, no flag will be set.

`options`

This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

Latexmk cleanup

Description

This rule calls the cleanup option of Latexmk, according to the provided parameters. All parameters are optional.

Syntax

```
% arara: lmkclean
```

Parameters

`include`
: This parameter, if equals to `all`, will remove all generated files, leaving only the source code intact; otherwise only the auxiliary files will be removed.

LuaLaTeX

Description

This rule maps LuaLaTeX, calling the `lualatex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: lualatex
```

Parameters

`action`
: This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`
: This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`
: This parameter is defined as boolean and sets the generation of SyncTeX data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`draft`
: This is a boolean parameter which sets the draft mode, that is, no PDF output is generated. When value set to true, the draft mode is enabled, while false disables it. If not defined, no flag will be set.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

Latexmk with LuaLaTeX

Description

This rule calls Latexmk with LuaLaTeX as engine. All parameters are optional.

Syntax

```
% arara: lualatexmk
```

`action`
: This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`
: This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`
: This parameter is defined as boolean and sets the generation of SyncTeX data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

`style`
: This parameter is used in case you want to provide a style for `makeindex`, if different than the default style.

LuaTeX

Description

This rule maps LuaTeX, calling the `luatex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: luatex
```

Parameters

`action`
: This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`
: This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`
: This parameter is defined as boolean and sets the generation of SyncTEX data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`draft`
: This is a boolean parameter which sets the draft mode, that is, no PDF output is generated. When value set to true, the draft mode is enabled, while false disables it. If not defined, no flag will be set.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

Make

Description

This rule maps Make, calling the `make` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: make
```

Parameters

`task`
: This parameter is used to set the task name for `make` to execute.

MakeGlossaries

Description

This rule maps MakeGlossaries, calling the `makeglossaries` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: makeglossaries
```

Parameters

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

MakeIndex

Description

This rule maps MakeIndex, calling the `makeindex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: makeindex
```

Parameters

`style`
: This parameter sets the index style. If not defined, `makeindex` relies on the default index style.

`german`
: This is a boolean parameter which sets the German word ordering in the index. If true, the German word ordering will be employed; if the value is set to false, `makeindex` will rely on the default behaviour.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

Nomencl

Description

This rule maps Nomencl, which is in fact a call to the `makeindex` command with the the nomenclature feature. All parameters are optional.

Syntax

```
% arara: nomencl
```

Parameters

`style`

This parameter sets the nomenclature style. If not defined, `makeindex` relies on the default nomenclature style.

`options`

This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

pdfLaTeX

Description

This rule maps pdfLaTeX, calling the `pdflatex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: pdflatex
```

Parameters

`action`

This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`

This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`

This parameter is defined as boolean and sets the generation of SyncTeX

data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`draft`
: This is a boolean parameter which sets the draft mode, that is, no PDF output is generated. When value set to true, the draft mode is enabled, while false disables it. If not defined, no flag will be set.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

Latexmk with pdfLaTeX

Description

This rule calls Latexmk with pdfLaTeX as engine. All parameters are optional.

Syntax

```
% arara: pdflatexmk
```

`action`
: This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`
: This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`
: This parameter is defined as boolean and sets the generation of SyncTeX data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

`style`
: This parameter is used in case you want to provide a style for `makeindex`, if different than the default style.

pdfTeX

Description

This rule maps pdfTeX, calling the `pdftex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: pdflatex
```

Parameters

`action`
: This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`
: This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`
: This parameter is defined as boolean and sets the generation of SyncTeX data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`draft`
: This is a boolean parameter which sets the draft mode, that is, no PDF output is generated. When value set to true, the draft mode is enabled, while false disables it. If not defined, no flag will be set.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

ps2pdf

Description

This rule maps pdf2pdf, calling the `ps2pdf` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: ps2pdf
```

Parameters

`output`
: This parameter is used to set the output PDF filename. If not provided, the default output name is set to `@{getBasename(file)}.pdf`.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

Sketch

Description

This rule maps Sketch, a small, simple system for producing line drawings of two or three-dimensional objects and scenes. All parameters are optional.

Syntax

```
% arara: sketch
```

Parameters

`input`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

songidx

Description

This rule maps `songidx`, a command line tool used to extract songs metadata from an file generated by the `songs` package[1] The parameter is mandatory.

Syntax

```
% arara: songidx
```

Parameters

`input`
: This parameter sets the name of the file generated by `songs` in which `songidx` will extract the songs metadata.

[1] `http://songs.sourceforge.net`, written by Kevin Hamlen.

TEX

Description

This rule maps TEX, calling the `tex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: pdflatex
```

Parameters

`action`

This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`

This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`options`

This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

XELATEX

Description

This rule maps XELATEX, calling the `xelatex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: xelatex
```

Parameters

`action`

This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`

This is a boolean parameter which sets the shell escape mode. If true,

shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`
: This parameter is defined as boolean and sets the generation of SyncTeX data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

Latexmk with XeLaTeX

Description

This rule calls Latexmk with XeLaTeX as engine. All parameters are optional.

Syntax

```
% arara: xelatexmk
```

`action`
: This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`
: This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`
: This parameter is defined as boolean and sets the generation of SyncTeX data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

`style`
: This parameter is used in case you want to provide a style for `makeindex`, if different than the default style.

XeTeX

Description

This rule maps XeTeX, calling the `xetex` command with the proper parameters, when available. All parameters are optional.

Syntax

```
% arara: xelatex
```

Parameters

`action`
: This parameter sets the interaction mode flag. Possible options are `batchmode`, `nonstopmode`, `scrollmode`, and `errorstopmode`. If not defined, no flag will be set.

`shell`
: This is a boolean parameter which sets the shell escape mode. If true, shell escape will be enabled; if the value is set to false, the feature will be completely disabled. If not defined, the default behaviour is rely on restricted mode.

`synctex`
: This parameter is defined as boolean and sets the generation of SyncTeX data for previewers. If true, data will be generated; false will disable this feature. If not defined, no flag will be set.

`options`
: This parameter is used to provide flags which were not mapped. It is recommended to enclose the value with single or double quotes.

8.2 Organizing directives

Actually, there's nothing much to say about directives, they are really easy to use. The important part when dealing with directives is to make sure we will only use the right amount of them. Remember, for each directive, there will be call to the command line tool, and this might take some time.

Since a directive can have as many parameters as its corresponding rule has, we need to take care. 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. 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.

Although arara reads the whole file looking for directives, it's a good idea to organize them at the top of the file. It will surely make your life easier, as you can quickly spot the compilation chain to be applied to the current document. If there's something wrong with a directive, don't worry, arara will be able to track the inconsistency down and warn us about it.

Chapter 9

Reference for rule library

Your brain may give birth to any technology, but other brains will decide whether the technology thrives. The number of possible technologies is infinite, and only a few pass this test of affinity with human nature.

Robert Wright

This chapter aims at discussing the reserved keywords of arara for directive arguments and special orb tags, their purpose and how to correctly use them in the context of a document.

9.1 Directive arguments

As seen in the previous chapters, arara has two reserved keywords for directive arguments which cannot be defined as arguments of a rule: `files` and `items`. Those variables do not hold a single value as the usual directive argument does, but they actually refer to a list of values instead. in the YAML format, a list is defined as a sequence of elements separated by a comma and enclosed with `[]`. For example, `items: [ a, b, c ]` is a list and refers to the elements `a`, `b` and `c`. Let's see in more details about each directive argument.

`files`
: When not defined with a proper value in the directive definition, `files`

contains only one value: the current file reference. When we explicitly add this argument to a directive, the value is overriden, and arara considers one iteration per element. In order words, if we have `foo: { files: [ a, b, c ] }`, arara will perform the execution of the task `foo` three times, one for each value of `files`. Each value of `files` is expanded to the `@{file}` orb tag in the rule context.

`items`
: The `items` directive argument, although it has the exact behaviour of `files` in the processing phase, happens to have a different semantics. Think of a rule that needs to process a list of elements, say, a list of extensions, files to copy, and so forth; for every value defined in `items`, arara will perform the execution the current task. It's important to note that `items` has an empty list by default. Each value of `items` is expanded to the `@{item}` orb tag in the rule context.

Both `files` and `items` can be used in any directive, if the rule of course makes use of their corresponding `@{file` and `@{item}` orb tags. Please note that, if those two lists are defined in the directive, arara will resolve the variables as the cartesian product of the lists.

9.2 Special orb tags

In the rule context, arara has four reserved keywords which cannot be assigned as arguments identifiers; each one of them has its own purpose and semantics, besides of course mapping different values. These orb tags are `@{file}`, `@{item}`, `@{parameters}` and `@{SystemUtils}`.

`file`
: This orb tags always resolve to the filename processed by arara. If the `files` directive argument is used in the directive definition, `@{file}` will resolve, in each iteration, to the current value of that list. The variable always hold an string value and it's never empty.

`item`
: The `@{item}` orb tag resolves to each element of the list of items defined through the `items` directive argument. In each iteration of the `items` list, `@{item}` will resolve to the current value of that list. The variable always hold an string value and it is empty by default.

`parameters`
: This orb tag is actually a map, that is, a collection of variables. The

`@{parameters}` orb tag is set with all the directive arguments and their corresponding values. The access to a variable is done through `parameters.<variable>`, so if we want to access the `foo` directive argument value, we simply write `@{parameters.foo}`.

`SystemUtils`
: This orb tag maps the `SystemUtils` class from the Apache Commons Lang library [1] and provides a lot of methods and properties in order to write cross-platform rules. Table 10.2 on page 100 presents the list of properties available in the `@{SystemUtils}` orb tag.

Since these are reserved keywords used for special orb tags, arara will raise an error if there's an attempt of assigning one of them as rule argument identifier.

References

[1] The Apache Software Foundation. *Apache Commons Lang*. 2001. URL: http://commons.apache.org/lang/ (cit. on p. 87).

Part III

For rulemakers

Chapter 10

Quick start

Cause and effect act in webs,
not chains.

Steve Grand

Now that we know about rules, directives and orb tags, it's time to come up with some examples. I know it might not be trivial to understand how arara works in a glance, 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 – actually, don't worry, arara has a card up its sleeve.

10.1 Writing rules

Before we proceed, I think it's important to mention this note again: we can't start values with `@` because this symbol is reserved for future use in the YAML format. For example, `foo: @bar` is an invalid YAML format, so the correct usage is to enclose it in quotes: `foo: '@bar'` or `foo: "@bar"`. We also need to enclose our strings with quotes in arara, but we can save them by simply adding the `<arara>` prefix to the value. In other words, `foo: <arara > @bar` is correctly parsed; when that keyword in that specific position is found, arara removes it.

Our first example is to add support to pdfLaTeX instead of using the default rule. Our first attempt to write this rule is presented in Code 39. Make sure to create a directory to store your own rules and don't forget to add this directory to the search path in the configuration file (Chapter 6).

Code 39: `pdflatex.yaml`, first attempt.

```
!config
identifier: pdflatex
name: PDFLaTeX
command: pdflatex "@{file}"
arguments: []
```

So far, so good. The `command` flag has the `pdflatex` program and the `@{file}` orb tag. Now we can add the `pdflatex` directive to our `.tex` file, as we can see in Code 40.

Code 40: `helloworld.tex`

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

\begin{document}
Hello world.
\end{document}
```

It's just a matter of calling `arara helloworld` – you can also provide the `.tex` extension by calling `arara helloworld.tex`, after all the extension will be removed anyway – and arara will process our file, according to the Code 41.

Code 41: arara output for the `pdflatex` task.

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

Running PDFLaTeX... SUCCESS
```

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` to call a package that makes use of it (for example, `minted`). It's very dangerous to enable shell escape globally, but changing the `pdflatex` call every time we need it sounds boring.

arara has a special treatment for cases like this. 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 42.

Code 42: `pdflatex.yaml`, second attempt.

```
1  !config
2  identifier: pdflatex
3  name: PDFLaTeX
4  command: pdflatex @{shell} "@{file}"
5  arguments:
6  - identifier: shell
7    flag: <arara> @{parameters.shell == "yes" ? "--shell-escape"
         : "--no-shell-escape" }
```

Line 7 from Code 42 makes use of the ternary operator `?:` which defines a conditional expression. In the first part of the evaluation, we check if `parameters.shell` 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, but arara has a clever way of doing that: a function to look for boolean values! In this case, we will use a function named `isTrue()`, available in the rule context. Please refer to Section 11.1 for a list of the available functions and their meanings. The new attempt is presented in Code 43

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.

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 once in a while. Code 44 shows our first attempt of writing this

Code 43: `pdflatex.yaml`, third attempt.

```
1  !config
2  identifier: pdflatex
3  name: PDFLaTeX
4  command: pdflatex @{shell} "@{file}"
5  arguments:
6  - identifier: shell
7    flag: <arara> @{ isTrue( parameters.shell, "--shell-escape" ,
         "--no-shell-escape" ) }
```

rule. Note that we are making use of another built-in function of arara named `getBasename()`; this function returns the name of the file without the extension.

Code 44: `makeindex.yaml`, first attempt.

```
1  !config
2  identifier: makeindex
3  name: MakeIndex
4  command: makeindex @{style} "@{ getBasename(file) }.idx"
5  arguments:
6  - identifier: style
7    flag: <arara> -s @{parameters.style}
```

As a follow-up to our first 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 enabling shell escape in the `pdflatex` rule from Code 43. The new rule is presented in Code 45.

The new `makeindex` rule presented in Code 45 looks good. We can now test the compilation workflow with an example. Consider a file named `helloindex.tex` which has a few index entries for testing purposes, presented in Code 46. 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 `arara helloindex.tex` in the terminal, we will obtain the same output from Code 47. The execution order is defined

Code 45: `makeindex.yaml`, second attempt.

```
!config
identifier: makeindex
name: MakeIndex
command: makeindex @{style} "@{ getBasename(file) }.idx"
arguments:
- identifier: style
  flag: <arara> -s @{parameters.style}
- identifier: german
  flag: <arara> @{ isTrue( parameters.german, "-g" ) }
```

Code 46: `helloindex.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 order of directives 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 a 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). arara relies on the Apache Commons Exec library to provide the system calls.

Code 47: 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 47, arara executed all the commands successfully. In Section 7.3 we discuss how arara works with commands and how to get their streams for a more detailed analysis.

For the next example, we will write a rule for both BibTeX and biber. Instead of writing two rules – one for each command – I'll show how we can use conditional expressions and run different commands in a single rule. The common scenario is to have each tool mapped to its own rule, but as we can see, rules are very flexible. Let's see how arara handles this unusual `bibliography` rule presented in Code 48.

The `bibliography` rule is quite simple, actually. If no `engine` is provided in the `bibliography` directive, the `default` element of the `engine` argument will be set to `bibtex`. Otherwise, if the `engine` parameter is set to `biber` – and only this value – the `engine` orb tag will expand the result to `biber`.

Code 48: `bibliography.yaml`

```
!config
identifier: bibliography
name: Bibliography
command: <arara> @{engine} @{args} @{ getBasename(file) }
arguments:
- identifier: engine
  flag: <arara> @{ isTrue( parameters.engine == "biber", "biber
      ", "bibtex" ) }
  default: bibtex
- identifier: args
  flag: <arara> @{parameters.args}
```

Code 49 presents only the header of our `biblio.tex` file using the new `bibliography` directive. Other options are shown in Table 10.1.

Code 49: `biblio.tex`

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

It's important to note that `bibtex` and `biber` differ in their flags, so I used a global `args` parameter. It is recommended to enclose the `args` value with single or double quotes. Use this parameter with great care, since the values differ from tool to tool. The output is presented in Code 50.

According to the terminal output shown in Code 50, arara executed all the commands successfully. A friendly warning: this rule is very powerful because of its flexibility, but the syntax – specially the conditional expression and the expansion tricks – might mislead the user. My advice is to exhaustively test the rules before deploying them into production. After all, better be safe than sorry.

Note that arara already includes both `bibtex` and `biber` rules. We believe this is the best approach to deal with such tools instead of a generic `bibliography` rule. Take a look on the existing rules, they might help the learning process.

Directive	Behaviour
`bibliography: { engine: bibtex }`	This directive sets the `engine` parameter to `bibtex`, which will expand the command to `bibtex` in the rule. Note that any value other than `biber` will expand the command to `bibtex`.
`bibliography: { engine: biber }`	This directive sets the `engine` parameter to `biber`, which will expand the command to `biber` in the rule. This is the only possible value that will set `biber` as the rule command.
`bibliography: { engine: bibtex, args: '-min-crossrefs=2' }`	This directive sets the `engine` parameter to `bibtex` and also provides an argument to the command. Note that the `args` value is specific to `bibtex` – using this argument value with `biber` will surely raise an error.
`bibliography: { engine: biber, args: '--sortcase=true' }`	This directive sets the `engine` parameter to `biber` and also provides an argument to the command. Note that the `args` value is specific to `biber` – using this argument value with `bibtex` will surely raise an error.

Table 10.1: Other directive options for `bibliography`.

Code 50: Running `biblio.tex`.

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

Running PDFLaTeX... SUCCESS
Running Bibliography... SUCCESS
Running PDFLaTeX... SUCCESS
```

10.2 Cross-platform rules

One of the goals when writing arara was to provide a cross-platform tool which behaves exactly the same on every single operating system. Similarly, the rules also follow the same idea, but sadly that's not always possible. After all, at some point, commands are bounded to the underlying operating system.

A rule that call `pdflatex`, for example, is easy to maintain; you just need to ensure there's an actual `pdflatex` command available in the operating system – in the worst case, arara warns about a nonexisting command. But there are cases in which you need to call system-specific commands. You could write two or three rules for the same task, say `makefoowin`, `makefoolinux`, and `makefoomac`, but this approach is not intuitive. Besides, if you share documents between operating systems, you'd have to also change the respective directive in your `.tex` file in order to reflect which operating system you are on.

Thankfully, there's a better solution for writing cross-platform rules which require system-specific commands. In Section 9.2, we mentioned about a special orb tag called `@{SystemUtils}` – it's now time to unveil its power. This orb tag is available for all rules and maps the `SystemUtils` class from the Apache Commons Lang library [1]. In other words, we have access to all methods and properties from that class.

Even though we have access to all public methods of the `SystemUtils` class, I believe we won't need to use them – the available properties are far more useful for us. Table 10.2 shows the most relevant properties for our context. The Apache Commons Lang documentation contains the full class description.

Every time we want to call any of the available properties presented in Table 10.2, we just need to use the `SystemUtils.PROPERTY` syntax, check the corresponding value through conditional expressions and define commands or arguments according to the underlying operating system.

Let's go back to our examples and add a new plain rule featuring the new `@{SystemUtils}` orb tag. Right after running `arara helloindex` successfully (Code 47), we now have as a result a new `helloindex.pdf` file, but also a lot of auxiliary files, as we can see in Code 51.

Code 51: List of all files after running `arara helloindex`.

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

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. For now, let's stick with a system-specific rule – don't worry, we will improve this rule later on.

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`. This special key is

Property	Description
IS_OS_AIX	True if this is AIX.
IS_OS_FREE_BSD	True if this is FreeBSD.
IS_OS_HP_UX	True if this is HP-UX.
IS_OS_IRIX	True if this is Irix.
IS_OS_LINUX	True if this is Linux.
IS_OS_MAC	True if this is Mac.
IS_OS_MAC_OSX	True if this is Mac.
IS_OS_NET_BSD	True if this is NetBSD.
IS_OS_OPEN_BSD	True if this is OpenBSD.
IS_OS_OS2	True if this is OS/2.
IS_OS_SOLARIS	True if this is Solaris.
IS_OS_SUN_OS	True if this is Sun OS.
IS_OS_UNIX	True if this is a Unix-like system, as in any of AIX, HP-UX, Irix, Linux, Mac OS X, Solaris or Sun OS.
IS_OS_WINDOWS	True if this is Windows.
IS_OS_WINDOWS_2000	True if this is Windows 2000.
IS_OS_WINDOWS_2003	True if this is Windows 2003.
IS_OS_WINDOWS_2008	True if this is Windows 2008.
IS_OS_WINDOWS_7	True if this is Windows 7.
IS_OS_WINDOWS_95	True if this is Windows 95.
IS_OS_WINDOWS_98	True if this is Windows 98.
IS_OS_WINDOWS_ME	True if this is Windows ME.
IS_OS_WINDOWS_NT	True if this is Windows NT.
IS_OS_WINDOWS_VISTA	True if this is Windows Vista.
IS_OS_WINDOWS_XP	True if this is Windows XP.

Table 10.2: Most relevant properties of `SystemUtils`.

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 52.

Code 52: `clean.yaml`, first attempt.

```
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 nonexisting file, `rm` will complain and return a value different than zero. This will make arara halt and print a big "failure" on screen, since a non-zero exit status is considered an abnormal execution. If we provide the `-f` flag, `rm` will not complain of a nonexisting file, so we won't be bothered for this trivial task.

Now we need to add the new `clean` directive to our `helloindex.tex` file (Code 46). Of course, `clean` will be the last directive, since it will only be reachable if everything executed before was returned withno errors. The new header of `helloindex.tex` is presented in Code 53.

Code 53: `helloindex.tex` with the new `clean` directive.

```
1 % arara: pdflatex
2 % arara: pdflatex
3 % arara: makeindex
4 % arara: pdflatex
5 % arara: clean: { files: [ helloindex.aux, helloindex.idx,
      helloindex.ilg, helloindex.ind, helloindex.log ] }
6 \documentclass{article}
7 ...
```

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. Time to run `arara helloindex` again and see if our new `clean` rule works! Code 54 shows both arara execution and directory listing. We expect to find only our source `helloindex.tex` and the resulting `helloindex.pdf` file.

Code 54: 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! But we have a big issue: if we try to use this rule in Windows, it doesn't work – after all, `rm` is not a proper Windows command. Worse, replacing `rm` by the equivalent `del` won't probably work. Commands like `del` must be called in the form `cmd /c del`. Should we write another system-specific rule, say, `cleanwin`? Of course not, there's a very elegant way to solve this issue: the `@{SystemUtils}` orb tag.

The idea is very simple: we check if arara is running in a Windows operating system; if true, we set the command to `cmd /c del`, or `rm -f` otherwise. The new version of our `clean` rule is presented in Code 55.

Code 55: `clean.yaml`, second attempt.

```
1 !config
2 identifier: clean
3 name: CleaningTool
4 command: <arara> @{ SystemUtils.IS_OS_WINDOWS ? "cmd /c del" :
     "rm -f" } "@{file}"
5 arguments: []
```

There we go, our first cross-platform rule! There's no need of writing a

bunch of system-specific rules; only one cross-platform rule is enough. We know that the `clean` rule will work as expected in every operating system, even if the task to be performed relies on system-specific commands. With cross-platform rules, we are able to write cleaner and more concise code.

There's another way of writing the `clean` rule, now with a built-in function instead of the `@{SystemUtils}` orb tag: we can use a function named `isWindows()` to check if arara is running in a Windows operating System. The third attempt of our `clean` rule is presented in Code 56.

Code 56: `clean.yaml`, third attempt.

```
1 !config
2 identifier: clean
3 name: CleaningTool
4 command: <arara> @{ isWindows( "cmd /c del" , "rm -f" ) } "@{
      file}"
5 arguments: []
```

Note that the `clean` rule is expecting `@{file}` to be overriden, since we rely on the reserved directive key `files`. If by any chance this rule is called without the `files` directive key, that is, an empty directive `% arara: clean`, I have very bad news to you: the rule will be expanded to `rm -f mydoc.tex` and your `.tex` file will be gone! Is there any way to avoid this behaviour? Yes, there is.

For our fourth attempt of rewritting the `clean` rule, we will make use of two new built-in functions. The first one is named `isFalse()`, which only expands the value if the conditional expression resolves to false; the second one is named `getOriginalFile()`, which holds the original reference to the file processed by arara. The idea here is very simple: if the current `@{file}` is different than the original file, run the task; otherwise, the whole command is expanded to an empty string – empty commands are discarded by arara. We will use a rule argument to hold the whole command, but note that the `flag` element is not important here, since we won't use this argument in the directive; only `default` matters in this context. The new `clean` rule is presented in Code 57.

Now we have a safe version of the `clean` rule. If we try to run arara on our document with `% arara: clean`, nothing will happen and our original file won't be removed. That means that `clean` will only take action when we have an explicit list of files to be removed, and even if the element in the `files` list is different than the original file.

Code 57: `clean.yaml`, fourth attempt.

```
1 !config
2 identifier: clean
3 name: CleaningTool
4 command: <arara> @{remove}
5 arguments:
6 - identifier: remove
7   default: <arara> @{ isFalse( file == getOriginalFile(),
        isWindows( "cmd /c del", "rm -f" ).concat(' "').concat(
        file).concat('"')) }
```

Take a look in all the default rules available in the project directory on GitHub. They are very easy to understand. If you get stuck in any part, a good advice is to enable the logging feature through the `--log` flag, since arara logs every expansion and command.

References

[1] The Apache Software Foundation. *Apache Commons Lang*. 2001. URL: http://commons.apache.org/lang/ (cit. on p. 99).

Chapter 11

Reference for rule library

I first saw the TEXbook lying beside a brand new Macintosh Plus back in 1985 and was instantly amazed by both.

Enrico Gregorio

This chapter presents a list of built-in functions of arara available in the rule context, as well as some notes on expansion. These functions have to be used always inside an orb tag, that is, `@{ <function> }`, in order to properly work, since they are written for the MVEL expression language.

11.1 Functions

arara features some functions in order to ease trivial tasks during the writing process of a rule. In this section, we will present a list of these functions, including their parameters and return value.

getOriginalFile

Syntax

```
string getOriginalFile()
```

Description

Returns the original file reference processed by arara as `string`.

isEmpty

Syntax

```
boolean isEmpty(string s)
```

Description

Checks if `s` is empty and returns a `boolean` value: `true` if `s` is empty, `false` otherwise.

Syntax

```
string isEmpty(string s1, string s2)
```

Description

Checks if `s1` is empty and returns a `string` value: `s2` if `s1` is empty, or an empty `string` otherwise.

Syntax

```
string isEmpty(string s1, string s2, string s3)
```

Description

Checks if `s1` is empty and returns a `string` value: `s2` if `s1` is empty, or `s3` otherwise.

isNotEmpty

Syntax

```
boolean isNotEmpty(string s)
```

Description

Checks if `s` is not empty and returns a `boolean` value: `true` if `s` is not empty, `false` otherwise.

Syntax

```
string isNotEmpty(string s1, string s2)
```

Description

Checks if `s1` is not empty and returns a `string` value: `s2` if `s1` is not empty, or an empty `string` otherwise.

Syntax

```
string isNotEmpty(string s1, string s2, string s3)
```

Description

Checks if `s1` is not empty and returns a `string` value: `s2` if `s1` is not empty, or `s3` otherwise.

isTrue

Syntax

```
boolean isTrue(string s)
```

Description

Checks if `s` has any of the values in the arara context that are considered `true` – `true`, `yes`, `y` and `1` – and returns a `boolean` value: `true` if `s` has a valid `true` value, or `false` otherwise.

Syntax

```
string isTrue(string s1, string s2)
```

Description

Checks if `s1` has any of the values in the arara context that are considered `true` – `true`, `yes`, `y` and `1` – and returns a `string` value: `s2` if `s1` has a valid `true` value, or an empty `string` otherwise.

Syntax

```
string isTrue(string s1, string s2, string s3)
```

Description

Checks if `s1` has any of the values in the arara context that are considered `true` – `true`, `yes`, `y` and `1` – and returns a `string` value: `s2` if `s1` has a valid `true` value, or `s3` otherwise.

Syntax

```
string isTrue(string s1, string s2, string s3, string s4)
```

Description

Checks if `s1` has any of the values in the arara context that are considered `true` – `true`, `yes`, `y` and `1` – and returns a `string` value: `s2` if `s1` has a valid `true` value, `s3` if `s1` has any of the values in the arara context that are considered `false` – `false`, `no`, `n` and `0` – or `s4` otherwise as a default fallback.

Syntax

```
string isTrue(boolean b, string s)
```

Description

Returns `s` if `b` is `true`, or an empty `string` otherwise.

Syntax

```
string isTrue(boolean b, string s1, string s2)
```

Description

Returns `s1` if `b` is `true`, or `s2` otherwise.

isFalse

Syntax

```
boolean isFalse(string s)
```

Description

Checks if `s` has any of the values in the arara context that are considered `false` – `false`, `no`, `n` and `0` – and returns a `boolean` value: `true` if `s` has a valid `false` value, or `false` otherwise.

Syntax

```
string isFalse(string s1, string s2)
```

Description

Checks if `s1` has any of the values in the arara context that are considered `false` – `false`, `no`, `n` and `0` – and returns a `string` value: `s2` if `s1` has a valid `false` value, or an empty `string` otherwise.

Syntax

```
string isFalse(string s1, string s2, string s3)
```

Description

Checks if `s1` has any of the values in the arara context that are considered `false` – `false`, `no`, `n` and `0` – and returns a `string` value: `s2` if `s1` has a valid `false` value, or `s3` otherwise.

Syntax

```
string isFalse(string s1, string s2, string s3, string s4)
```

Description

Checks if `s1` has any of the values in the arara context that are considered `false` – `false`, `no`, `n` and `0` – and returns a `string` value: `s2` if `s1` has a valid `false` value, `s3` if `s1` has any of the values in the arara context that are considered `true` – `true`, `yes`, `y` and `1` – or `s4` otherwise as a default fallback.

Syntax

```
string isFalse(boolean b, string s)
```

Description

Returns `s` if `b` is `false`, or an empty `string` otherwise.

Syntax

```
string isFalse(boolean b, string s1, string s2)
```

Description

Returns `s1` if `b` is `false`, or `s2` otherwise.

trimSpaces

Syntax

```
string trimSpaces(string s)
```

Description

Returns `s` with the trailing and leading spaces trimmed.

getFilename

Syntax

```
string getFilename(string s)
```

Description

This function takes a file path in the form of a `string` and returns a `string` containing only the file name, or an empty `string` in case of error.

getBasename

Syntax

```
string getBasename(string s)
```

Description

Returns the base name of `s` as a `string`, that is, the file name without the extension, or an empty `string` in case of error.

getFiletype

Syntax

```
string getFiletype(string s)
```

Description

Returns the file type of `s` as a `string`, that is, the extension of the file name, or an empty `string` in case of error.

getDirname

Syntax

```
string getDirname(string s)
```

Description

This function takes a file path in the form of a `string` and returns a `string` containing only the directory structure without the file name, or an empty `string` in case of error.

isFile

Syntax

```
boolean isFile(string s)
```

Description

Returns `true` if `s` is a valid reference to a file, or `false` otherwise.

isDir

Syntax

```
boolean isDir(string s)
```

Description

Returns `true` if `s` is a valid reference to a directory, or `false` otherwise.

isWindows

Syntax

```
string isWindows(string s1, string s2)
```

Description

Returns `s1` if arara is running in a Windows operating system, or `s2` otherwise.

isLinux

Syntax

```
string isLinux(string s1, string s2)
```

Description

Returns `s1` if arara is running in a Linux operating system, or `s2` otherwise.

isUnix

Syntax

```
string isUnix(string s1, string s2)
```

Description

Returns `s1` if arara is running in a Unix operating system, or `s2` otherwise.

isMac

Syntax

```
string isMac(string s1, string s2)
```

Description

Returns `s1` if arara is running in a Mac operating system, or `s2` otherwise.

All the functions described are available in the rule context and can be concatenated in order to create more complex checkings.

11.2 Notes on expansion

It's important to observe that arara always try to rely on a smooth fallback to empty strings in case of function errors or unused arguments. This approach allows the application to not halt in case of a recoverable situation. If, for example, arara finds an empty command to execute – like in the `clean` rule presented in Code 57 when `files` isn't used – the task is simply ignored. That way, we can make more robust rules without worrying too much with expansion.

Remember that arara basically deals with string values, and some times, with boolean operations. We decided to stick with those two types because of simplicity. Taking care of string comparisons, using the built-in functions and limiting the scope of the command is sufficient to write good rules.