Easy TEX automation with arara

Paulo Roberto Massa Cereda

Abstract

This article introduces **arara**, a TEX automation tool based on rules and directives. The tool is an effort to provide a concise way to automate the daily TEX workflow for users and also package writers. The author presents a quick tutorial on how **arara** works and unveils some of the new features and improvements of the upcoming version.

Sommario

Questo articolo introduce **arara**, uno strumento di automazione di TEX basato su regole e direttive. Lo strumento risulta da uno sforzo di fornire un modo conciso per automatizzare la procedura di lavoro quotidiana degli utenti e dei programmatori dei pacchetti. L'autore presenta un rapido corso sul funzionamento di **arara** e svela alcune delle nuove caratteristiche e miglioramenti della versione in arrivo.

1 Prologue

Back in 2007, I had a complex TEX project to tackle. Needless to say, I was very worried, not only because little I knew about the ways of the TEX force, but the compilation steps required to achieve the final results were getting out of hand. Something had to be done, or this whole adventure was about to end in the worst possible way.

I tried `latexmk` first and it did a pretty good job. Sadly, my project had indeed a very complex compilation workflow and the tool was being pushed to its limits – my `.latexmkrc` file was getting more and more complicated as time went by. I then thought I should try a different solution.

1.1 Beware of lightning bolts

Suddenly, it was like being struck by a lightning bolt: I *knew* all the steps I had to reproduce beforehand, I only had to find a way to automate them! Trying to infer things from (at least) context-sensitive languages like TEX would mean a lot of work! And all that guessing was unnecessary energy waste (although it could be a great excuse for eating more chocolate). So let us take the low resistance route.

So far, so good. I knew what I had to do, but how tell that to my potential new tool? Looking at how a compiler works, i.e. read a source file, ignore all comments and processes the rest, I thought to exploit the complement of it? My new tool would read instructions inside TEX comments, which would be ignored by conventional engines, thus causing no harm and side effects at all!

FIGURE 1: Do you like araras? We do, specially our tool which shares the same name of this colorful bird.

Time to make things happen. I sat in front of my computer and started to code while listening to Pink Floyd. In a couple of hours, I had a new tool tackling my TEX project. But the adventure was not over.

1.2 I blame Enrico and Marco

I mentioned my typographic adventure in the chat room of the TEX community at StackExchange and my good friend Enrico Gregorio encouraged me to release a public version of my newly created tool. So **arara** was introduced to the TEX world. Figure 1 shows a lovely photo of the bird which inspired my tool's name.

The reception was surprisingly positive. Marco Daniel, a German TEXnician and a good friend, liked the idea so much and became an evangelist. He helped me develop **arara**, writing code and providing feedback.

But I was not ready for a major release. I am far from being a perfectionist, but the code at the time was not acceptable yet. **arara** had just hatched (version 1.0) and required some code improvements before taking any further steps.

1.3 I blame Brent

It took me a lot of versions. When the counter stopped at version 3.0, Brent Longborough, `memoir` fanboy and a good friend, helped me with the code and the user manual. Then we decided it was time for **arara** to graduate and be released in TEX Live. My life had changed.

It was a success. A lot of people liked the idea of explicitly telling **arara** how to compile their documents instead of relying on guesswork. But then, the inevitable happened: a lot of bugs had emerged from the dark depths of my code.

1.4 I blame Nicola

I was about to give up. My code was not awful, but there were a couple of critical and blocking bugs. Something very drastic had to be done in order to put arara back on track. I decided to rewrite the tool entirely from scratch. I created a sandbox and started working on the new code.

My good friend Nicola Talbot helped me with the new version, writing code, fixing bugs and suggesting new features. Soon, we all reached a very pleasant result. It was like arara was about to hatch again. Version 4.0 was at our hands.

1.5 The future

Sadly, version 4.0 is not yet in TEX Live, and the reason for that is quite embarassing: I had no time to work on the new user manual yet! Since the new 4.0 version ships with several features, improvements and bug fixes, it will take some time to cover them all. Hopefully, I will manage to finish this manual by the end of the year, so the new shiny version will soon make our users happy.

This article covers some new features and improvements of version 4.0 of arara. Of course, it would take more than a couple of pages to properly cover the tool usage, but at least this text will provide some insights of things to come. Thank you very much, dear reader, for joining my adventure. Have a glass of wine and enjoy the reading.

2 Two-minute tutorial

This section is a quick walkthrough on how arara works. As I have mentioned in the prologue, the tool takes a different approach compared to other tools on how to inspect TEX code, and the key difference is that there is no guesswork. Let us say we have the following file `hello.tex` as an example:

```
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

Now, when running either `latexmk hello` or `rubber -pdf hello` in the terminal, we will get the expected result, but `arara hello` will not produce anything! In fact, arara will even complain about it:

```
It looks like no directives were
found in the provided file. Make
sure to include at least one
directive and try again.
```

What a rude bird! Wait a second, in fact, the tool is actually telling us what happened: there were no directives.

2.1 Directives

A directive is a special comment inserted in the source file in which you indicate how arara should behave. You can insert as many directives as you want and in any position of the file. Let us take a look at how a directive looks like:

`% arara:` ⟨*directive*⟩

That looks interesting: so I can replace ⟨*directive*⟩ by any command I want, right? Suppose I want to run a command named `foo` on my previous file, so let us try:

```
% arara: foo
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

Surprisingly, arara will complain once again! The tool will always try to help us and describe exactly what has happened:

```
I could not find a rule named
'foo' in the provided rule paths.
Perhaps a misspelled word? I
was looking for a file named
'foo.yaml' in the following
paths in order of priority:
(/home/paulo/arara/rules)
```

According to the message, arara was not looking for a command, it was looking for a rule! This bird is surely naughty.

2.2 Rules

It is important to note that a directive is not the command to be executed, but the name of the rule associated with that directive! So there has to be a rule named `foo` in order for arara to understand what to do (once a directive is found, the tool will look for the associated rule).

Fortunately, arara is released with several rules out of the box, so it covers most common use cases. Should you wish to define your own rule, the user manual has an entire chapter dedicated to how to write them. In our previous example, we would have to come up with our own `foo` rule, but that is a story for another day (and another article).

Back to our document, how do we compile `hello.tex` with pdfLATEX? It is just a matter of including the right name of the rule to be invoked (in our case, `pdflatex`):

```
% arara: pdflatex
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

Now, after running `arara hello`, we will finally get the expected result, the very same we got from `latexmk hello` or `rubber -pdf hello`. Hooray!

2.3 Types of directives

Now that we have learned about rules and directives, it is time to dive deeper into the directive concept. In short, there are two types of directives in **arara**. The first one has already been mentioned, it has only the rule name (identifier):

`% arara:` ⟨*directive*⟩

Sometimes, however, we need to provide additional information to the rule. That is reason for the second type, the parametrized directive, to exist.

As the name indicates, we have directive arguments! They are mapped by their identifiers and not by their positions. The syntax for a parametrized directive is:

`% arara:` ⟨*directive*⟩`: {` ⟨*arguments*⟩ `}`

Each argument is defined according to the rule mapped by the directive. This means you cannot use an argument `foo` in a directive `bar` if the rule `bar` does not offer support for it (that is, `bar` has to have `foo` defined as argument in its list of arguments inside the rule scope). The syntax for an argument is:

⟨*key*⟩ `:` ⟨*value*⟩

Suppose we would like to enable shell escape for PDFLaTeX when compiling `hello.tex`. We can achieve that by providing a parametrized directive, like this one:

```
% arara: pdflatex: { shell: yes }
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

Of course, the `shell` argument is defined in the `pdflatex` rule scope, otherwise **arara** would raise an error about an invalid argument key. If we try to inject a `foo` argument in the previous parametrized directive, we will get this message:

```
I have spotted an error in
rule 'pdflatex' located at
'/home/paulo/arara/rules'. I
found these unknown keys in the
directive: '(foo)'. This should
be an easy fix, just remove them
from your map.
```

As the message suggests, we need to remove the unknown argument key from our directive or rewrite the rule in order to include it. The first option is, of course, easier.

2.4 Overriding the default document

There is a reserved argument key named `file`, whose value is a list (represented by square brackets), which is available for every rule. If you want to override the default value of the main document for a specific directive, use this argument key on that directive and assign a list to it. For example, you if you want to run PDFLaTeX on files `foo.tex` and `bar.tex` instead of the default `hello.tex`, we can use the following trick (there is a new feature in this code, but I will talk about it later on, in Subsection 3.5):

```
% arara: pdflatex: {
% arara: --> files: [ foo.tex, bar.tex ]
% arara: --> }
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

Note that **arara** will handle this occurrence as two invocations of PDFLaTeX, one for each provided file (the terminal output and log file will tell you so).

2.5 Ready, set, go!

I believe the workflow is pretty straightforward. Of course, there is much more to **arara** than what I have described above, which covered enough to understand the basics and safely move on to the new features of version 4.0 and have fun. The user interface is one of the major selling points of **arara**, as it provides a clean and easy pattern for all users, from newbies to TeXnicians.

3 New features and improvements

This section unveils some of the new features and improvements in version 4.0. As mentioned in Section 2, the tool usage remains the same, so do not worry with incompatibilities (apart from just one change for rule makers in the rule scope, but the fix is incredibly easy and with no impact whatsoever to the casual users).

3.1 Friendly and helpful messages

As noted in Section 2, messages were developed to be friendly and helpful. In version 4.0, we decided to make them even friendlier and include suggestions for correcting errors or improving usage. For, example, consider the following message:

```
I could not find a rule named
'foo' in the provided rule paths.
Perhaps a misspelled word? I
was looking for a file named
'foo.yaml' in the following paths
in order of priority:
(/home/paulo/arara/rules)
```

The message does not only tells you that the `foo` rule was not found, but it also provides information about all the search locations. This bird is actually quite clever! The next message is about an unknown argument key:

```
I have spotted an error in
rule 'pdflatex' located at
'/home/paulo/arara/rules'. I
found these unknown keys in the
directive: '(foo)'. This should
be an easy fix, just remove them
from your map.
```

This message tells us the rule, its location and a set of unknown argument keys. It also provides a suggestion as how to fix this issue, which involves removing the problematic argument from the directive map. Let us see yet another message:

```
I read a directive (1) and found
out that 'files' requires a list.
Please make sure to correct the
type to a proper list and try
again.
```

The message tells us that `files` was used as a parameter in the directive found in line 1 of the source code, but the type is not a list. The message also suggests how to fix this issue.

3.2 Improved command line layout

The command line layout was completely redesigned in order to look more pleasant to the eye (after all, we work with TEX and friends). First and foremost, arara displays details about the file being processed, including size and modification status:

```
Processing 'hello.tex' (size: 132
bytes, last modified: 08/21/2015
11:09:04), please wait.
```

The list of tasks was also redesigned to be fully justified, and each entry displays both task and subtask names, besides of course the usual execution result. Figure 2 shows the new output. Note that arara also displays the execution time before terminating.

Verbose mode was also redesigned in order to avoid unnecessary clutter (there would be a lot of information already on screen), so it would be easier to spot each task. Error messages are now displayed in a box of their own, separated from the list of tasks.

3.3 Improved logging

The logging framework was also improved in order to better organize gathered information. The `arara.log` file contains the following enhancements (logging can be enabled through the `--log` flag):

1. System information is automatically collected in order to help debugging, such as the underlying operating system, architecture, Java virtual machine version and path, as well as the arara executable path.

2. Details about the file being processed, including size and modification status, are added to the logging handler (the very same information displayed in the terminal output).

3. The tool now includes all potential patterns found during the directive extraction phase, including the line number and content.

4. All directives, after being properly validated, are listed in the registry, including their potential arguments and conditionals.

5. Every task and subtask have an entry in the registry, containing the rule identifier and path, the corresponding system commands and output buffers, as well as the result status.

The resulting file is less cluttered and more useful when it comes to monitor and debug the automation process (it also helps me to guide users towards solving potential usage problems with arara).

3.4 Improved workflow

In previous versions, the workflow consisted of three steps: extract the directives from the source code, evaluate them beforehand and then execute all directives in batch mode. Starting with arara 4.0, we change this workflow to be based more on the REPL concept (read, eval, print, loop). In other words, directives are still extracted beforehand, but the tool will evaluate and execute one directive at a time before moving to the next one in the list.

This subtle change allows rules to be more context-aware, in the sense of inspecting the environment after the previous task has finished and take the appropriate actions, if needed.

3.5 Multiline directives

Sometimes, directives can span several columns of a line, particularly the ones with several arguments or conditionals. From arara 4.0, we can split a directive into multiple lines by adding `% arara: -->` to each line which should compose the directive (there is no need of them to be in contiguous lines, though):

```
% arara: pdflatex: {
% arara: --> shell: yes,
% arara: --> synctex: yes
% arara: --> }
```

It is worth mentioning that multiline support works with conditionals as well (described later on, in Subsection 3.9):

```
  __ _ _ __ __ _ _ __ __ _
 / _‘ | ’__/ _‘ | ’__/ _‘ |
| (_| | | | (_| | | | (_| |
 \__,_|_|  \__,_|_|  \__,_|

Processing ’hello.tex’ (size: 132 bytes, last modified:
08/21/2015 11:09:04), please wait.

(PDFLaTeX) PDFLaTeX engine ............................. SUCCESS
(Clean) Cleaning feature ............................... SUCCESS
(Clean) Cleaning feature ............................... SUCCESS

Total: 0.32 seconds
```

FIGURE 2: Improved command line layout in arara.

```
% arara: pdflatex
% arara: --> if missing(’pdf’)
% arara: --> || changed(’tex’)
```

3.6 --dry-run mode

Another interesting feature is the inclusion of a `--dry-run` flag, which goes through all the work of reading the tasks and subtasks, but without actually executing them. For example, `arara hello -n` (the long option `--dry-run` also works) will display the following result:

```
[DR] (PDFLaTeX) PDFLaTeX engine
----------------------------------------
Authors: Marco Daniel, Paulo Cereda
About to run: [ pdflatex, hello.tex ]
```

Note that the rule authors are displayed (so they can be blamed in case anything goes wrong), as well as the system command to be executed. It is an interesting approach to see everything that will happen to your document and in which order. It is important to observe, though, that conditionals are not evaluated in this mode.

3.7 Support for lists and maps as values

In previous versions, only `files` could be a list (Subsection 2.4). From version 4.0 on, directive arguments can hold any type of value, from single elements to maps and lists (of course, the rule has to be prepared to handle a list or a map, or even a combination of both). For example, the new `clean` rule allows us to specify a list of extensions to be removed (Figure 2 also shows the output of this execution):

```
% arara: pdflatex
% arara: clean: {
% arara: --> extensions: [ aux, log ]
% arara: --> }
\documentclass{article}
\begin{document}
Hello world!
\end{document}
```

There are several helper methods available in the rule scope to ease the use of lists and maps, so rule makers can have fun with these data types.

3.8 File hashing

arara 4.0 features four methods for file hashing in the rule and directive scopes, presented as follows:

- `changed(extension)`: checks if the file's basename concatenated with the provided extension has changed its checksum from last verification.
- `changed(file)`: the very same idea as the previous method, but with a proper `File` object instead.
- `unchanged(extension)`: checks if the file's basename concatenated with the provided extension is unchanged from last verification. It is the opposite of the `changed(...)` method.
- `unchanged(file)`: the very same idea as the previous method, but with a proper `File` object instead.

The value is stored in a file named `arara.xml` as a pair containing the file path and its corresponding CRC-32 hash (the file is created if missing). If the entry already exists, the value is updated, or created otherwise.

3.9 Conditionals

One of the most awaited features that version 4.0 introduces is the support of conditionals, that is, logic expressions processed at runtime in order to determine whether and how the directive should be processed. The following types are allowed:

- `if`: evaluated beforehand, the directive is interpreted if and only if the result is true.

- `unless`: evaluated beforehand, the directive is interpreted if and only if the result is false.
- `until`: directive is interpreted the first time, then the evaluation is done; while the result is false, the directive is interpreted again and again.
- `while`: evaluated beforehand, the directive is interpreted if and only if the result is true, and the process is repeated while the result still holds true.

It is important to observe that arara has a maximum number of loops, that is, we do not want a directive to be executed forever. The default value is set to 10, but it can be overriden either in the configuration file (through the `loops` key) or in the command line flag (through the `--max-loops` flag).

Several methods are available in the directive scope in order to ease the writing of conditionals, including (file hashing methods are available as well):

- `exists(extension)`: checks whether the file's basename concatenated with the provided extension exists. For example, if the file is named `foo.tex`, `exists('pdf')` will check whether `foo.pdf` exists.
- `exists(file)`: checks whether the file exists. Note that this is a proper call to the `File` object.
- `missing(extension)`: the opposite of exists, checks whether the file's basename concatenated with the provided extension does not exist.
- `missing(file)`: checks whether the file does not exist. As mentioned before, this is a proper call to the `File` object.
- `found(extension, regex)`: checks whether the file's basename concatenated with the provided extension matches the provided regular expression.
- `found(file, regex)`: the very same idea as the previous method, but with a proper `File` object instead.

Logical connectors can be used in order to create boolean expressions: `&&` (and), `||` (or), and `!` (negation). The following example runs PDFLaTeX if there is no resulting PDF file or if the source code has changed from last execution:

```
% arara: pdflatex
% arara: --> if missing('pdf')
% arara: --> || changed('tex')
```

The next example tells arara to run PDFLaTeX until the corresponding log file does not contain undefined references anymore:

```
% arara: pdflatex
% arara: --> until !found('log',
% arara: --> 'undefined references')
```

As mentioned, a lot of methods are available in the directive scope in order to provide a better experience and consistent execution, but I cannot list them all here. Please refer to the user manual and check the complete reference (when, of course, I finally manage to finish it).

4 Closing remarks

It took me so long to finally fix bugs and implement new features and enhancements, but I strongly believe the final result made the wait truly worthwhile. arara has become mature with time, and so has the code base. It is wonderful to look back at when I started and where we are today. I owe everything to my team and also to the hundreds of friends out there who decided to give this humble tool a try. Thank you very much.

arara is already avaliable in TEX Live and also as a standalone tool, but note that the version covered in this article is not officially released yet. The tutorial presented in Section 2 still applies to version 3.0, but the new features and improvements presented in Section 3 really require at least version 4.0, which is yet only available in my source code repository:

https://github.com/cereda/arara

Hopefully, I will soon be able to finish the user manual and finally release version 4.0 to the entire TEX world. Feel free to contribute to the project by submitting bugs, sending pull requests or maybe by translating the tool to your language. And if you want to support the LaTeX development with a donation, the best way to do this is by donating to your local TEX Users Group.

Happy TEXing with arara 4.0!

Acknowledgments

The author wishes to thank Enrico Gregorio, Francesco Endrici and all friends from GUIT for the opportunity of writing this humble article for ArsTEXnica.

▷ Paulo Roberto Massa Cereda
San Paolo University, Brasil
paulo.cereda@usp.br