A rollback concept for packages and classes

Frank Mittelbach

Abstract

In 2015 a rollback concept for the LaTeX kernel was introduced. Providing this feature allowed us to make corrections to the software (which more or less didn't happen for nearly two decades) while continuing to maintain backward compatibility to the highest degree.

In this paper we explain how we have now extended this concept to the world of packages and classes, which was not covered initially. As classes and extension packages have different requirements compared to the kernel, the approach is different (and simplified). This should make it easy for package developers to apply it to their packages and authors to use when necessary.

Contents

1 Introduction

In 2015 we introduced a rollback concept for the LaTeX kernel that enables a user to request a kernel rollback to its state at a given date by using the latexrelease package [1]. For example,

```
\RequirePackage[2016-01-01]{latexrelease}
```

would result in undoing all kernel modifications (corrections or extensions) released between the first of January 2016 and the current date.[1] Undoing means reinstalling the definitions current at the requested date and normally also removing new commands from TeX's memory so that `\newcommand` and similar declarations do not fall over because a name is already declared.

This mechanism helps in correctly processing older documents that contain workarounds for issues with an older kernel, issues that have since been fixed in a way that would make the old document fail, or produce different output, when processed with the newer, fixed kernel.

If necessary, the latexrelease package also allows for rolling the kernel forward without installing a new format. For example, if the current installation is dated 2016-04-01 but you have a document that requires a kernel with date 2018-01-01, then this can be achieved by starting it with

```
\RequirePackage[2018-01-01]{latexrelease}
```

provided you have a version of the latexrelease package that knows about the kernel changes between the date of your kernel and the requested date. Getting this version of the package is simple as the latest version can always be downloaded from CTAN. Thus you will be able to process your document correctly even when updating your complete installation is not advisable or impossible for one or another reason.

However, rolling back the kernel state is only doing half of the job: the LaTeX universe consists of many add-on packages and those were not affected by a kernel rollback request. We are therefore now extending the concept by providing a much simpler method for use in packages and classes, one that we think will be straightforward for developers and also easy for document authors to use.

Unlike the method used by the kernel, which tracks every change individually and is able to roll back the code to precisely the state it had on any given day, the new method for packages and classes is intended to cover only major change points, e.g., the introduction of major new features or (incompatible) changes in syntax or interfaces.

As we will have only a few rollback points per package or class, the different releases are all stored in separate files. In the main file it therefore only needs a single declaration per release to enable rollback. The downside is, of course, that for each release the whole package code is stored, instead of managing the differences between releases. This is one of the

[1] There are a few exceptions as some modifications are kept: for example, the ability to accept date strings in ISO format (e.g., 2016-01-01) in addition to the older LaTeX convention (e.g., 2016/01/01). These are not rolled back because removing such a feature would result in unnecessary failures.

reasons why this approach should be used only for major changes, i.e., at most a handful in the lifetime of a package.

From a technical perspective it is also possible to use the method introduced with latexrelease in package and class files, i.e., to mark up modifications using the commands `\IncludeInRelease` and `\EndIncludeInRelease` — the package's documentation [1] gives some advice on how to apply it in a package scenario — but the use of these commands in package code is cumbersome and results in fairly unreadable code, especially when there are many minor changes. This is an acceptable price to pay for fairly stable code, such as the kernel itself, since it offers complete control over the rollback to any date, but it is not truly practical in package or class development and so, to our knowledge, it has therefore never been used up to now. Section 5.4 gives some advice on how to achieve fine-grain control in a somewhat simpler manner.

2 Typical scenarios

A typical example, for which such a rollback functionality would have provided a major benefit (and will do so for packages in the future), is the caption package by Axel Sommerfeldt. This package started out under the name of caption with a certain user interface. Over time it became clear that there were some deficiencies in the user interface; to rectify these without making older documents fail, Axel introduced caption2. At a later point the syntax of that package itself was superseded, resulting in caption3 and then, finally, that got renamed back to caption. So now older documents using caption will fail whilst documents from the intermediate period will require caption2 (which is listed as superseded on CTAN but is still distributed in the major distributions). So users accustomed to copying their document preamble from one document to the next are probably still continuing to use it without noticing that they are in fact using a version with defective and limited interfaces.

Another example is the fixltx2e package that for many years contained fixes to the LaTeX kernel. In 2015 these were integrated into the kernel so that today this package is an empty shell, only telling the user that it is no longer needed. However, if you process an old document (from before 2015) that loads fixltx2e then of course the fixes originally provided by this package (like the corrections to the floats algorithm) would get lost as they are now neither in the kernel nor in the "empty" fixltx2e package if that doesn't roll back as well — fortunately it does and always did, so in reality it isn't quite an empty shell.

A somewhat different example is the amsmath package, which for nearly a decade didn't see any corrections even though several problems have been found in it over the years. If such bugs finally get corrected, then that would affect many of the documents written since 2000, since their authors may have manually worked around one or another of the deficiencies. Of course, as with the caption package, one could introduce an amsmath2, amsmath3, ... package, but that puts the burden on the user to always select the latest version (instead of automatically using the latest version unless an earlier one is really needed).

3 The document interface

By default LaTeX will automatically use the current version of any class or package — and prior to offering the new rollback concept it always did that unless the package or class had its own scheme for providing versioning, either using alternative names or by hand-coded options to select a version.

3.1 Global rollback

With the new rollback concept all the user has to do (if he or she wants their document processed with a specific version of the kernel and packages) is to add the latexrelease package at the beginning of the document and specify a desired date as the package option, e.g., just as in the first example:

```
\RequirePackage[2018-01-01]{latexrelease}
```

This will roll back the kernel to its state on that day (as described earlier) and for each package and the document class it will check if there are alternate releases available and select the most appropriate release of that package or class in relation to the given date.

3.2 Individual rollback

There is further fine-grain adjustment possible: both `\documentclass` as well as `\usepackage` have a second (little known) optional argument that up to now was used to allow the specification of a "minimal date". For example, by declaring

```
\usepackage[colaction]
           {multicol}[2018-01-01]
```

you specify that multicol is expected to be no older than the beginning of 2018. If only an older version is found, then processing such a document results in a warning message:

```
LaTeX Warning: You have requested, on input
   line 12, version '2018-01-01' of package
   multicol, but only version
   '2017/04/11 v1.8q multicolumn formatting
   (FMi)' is available.
```

The idea behind this approach is that packages seldom change syntax in an incompatible way, but more often add new features: with such a declaration you can indicate that you need a version that provides certain new features.

The new rollback concept now extends the use of this optional argument by letting you additionally supply a target date for the rollback. This is done by prefixing a date string with an equal sign. For example,

```
\usepackage{multicol}[=2017-06-01]
```

would request a release of multicol that corresponds to its version in June 2017.

So assuming that at some point in the future there is a major rewrite of this package that changes the way columns are balanced, the above would request a fallback to what right now is the current version from 2017-04-11. The old use of this optional argument is still available because presence or absence of the = determines how the date will be interpreted.

The same mechanism is available for document classes via the `\documentclass` declaration, and for `\RequirePackage` if that is ever needed.

3.3 Specifying a version instead of a date

Specifying a rollback date is most appropriate if you want to ensure that the behavior of the processing engine (i.e., the kernel and all packages) corresponds to that specific date. In fact, once you are finished with editing a document, you can preserve it for posterity by adding this line:

```
\RequirePackage[⟨today's-date⟩]
               {latexrelease}
```

This would mean that it will be processed a little more slowly (since the kernel may get rolled back and each package gets checked for alternate versions), but it would have the advantage that processing it a long time in the future will probably still work without the need to add that line later.

However, in a case such as the caption package or, say, the longtable package, that might eventually see a major new release after several years, it would be nice to allow the specification of a "named" release instead of a date: for example, a user might want to explicitly use version 4 rather than 5 of longtable when these versions have incompatible syntax, or produce different results.

This is also now possible if the developer declares "named" releases for a package or class: one can then request a named version simply by using this second optional argument with the "name" prefixed by an equal sign. For example, if there is a new version of longtable and the old (now current) version is labeled "`v4`", then all that is necessary to select that old version is

```
\usepackage{longtable}[=v4]
```

Note that there is no need to know that the new version is dated 2018-04-01 (nor to request a date before that) to get the old version back.

The version "name" is an arbitrary string at the discretion of the package author — but note that it must not resemble a date specification, i.e., it must not contain hyphens or slashes, since these will confuse the parsing routine.[2]

3.4 Erroneous input

The user interface is fairly simple and to keep the processing speed high the syntax checking is therefore rather light. Basically the standard date parsing from the kernel is used, which is rather unforgiving if it finds unexpected data.

Basically any string containing a hyphen or a slash will trigger the date parsing which then expects two hyphens (for the case of an ISO date) or two slashes (otherwise) and other than these separators, only digits. If it does find anything else, chances are that you will get a "`Missing \begin{document}`" error or, perhaps even more puzzling, a strange selection being made. For example, `2011/02` may mean to us February 2011 but for the parsing routine it is some day in the year 20 A.D. That is, it gets converted to the single number `201102`, so that, when this number is compared numerically to, say, `20000101`, it will be the smaller number, i.e., earlier, even though the latter is the numerical representation of January 1st 2000.

So, bottom line: do not misspell your dates and all is fine. That hasn't been a problem in the past, so hopefully it will be okay to continue with just this light checking. If not, then we may have to extend the checks made during parsing.

3.5 Advice for early adopters

If your document makes use of the new global rollback features, then it should be processable at any installation later than early 2015, when the latexrelease package was first introduced. If the installation is even older, then it needs upgrading or, at least, one has to add a current latexrelease package to the installation.

However, if your document uses the new concept for individual rollbacks of packages or classes (i.e.,

[2] Of course more sophisticated parsing could fix this, but we use fast and simple parsing that scans for slashes or hyphens with no further analysis.

with the =... syntax in the optional argument), then it is essential to use a LaTeX distribution from 2018 or later.[3] Earlier distributions will choke on the equal sign inside the argument as they will only expect to see a date specification there.

4 The package/class interface

The rollback mechanism for packages or classes is provided by putting, at the beginning of the file containing the code, a declaration section that informs the kernel about existing alternative releases.

These declarations have to come first and have to be ordered by date because the loading mechanism will evaluate them one by one and, once a suitable release is found, it will be loaded and then processing of the main package or class file will end. If there are no such declarations, or if the older releases are all ruled out for one reason or another, processing will continue as normal by reading all of the main file.

The old releases are stored in separate files, one for each release, and we suggest using a scheme such as ⟨*package-name*⟩-⟨*date*⟩.sty as this is easy to understand and will sort nicely within a directory. However, any other scheme will do as well, as the name is part of the declaration.

The contents of this release file is simply the package or class file as used in the past. This means that before making a new version all you need to do is to make a verbatim copy of the current file and give it a new suitable name.[4]

This way it is also straightforward to include older releases after the fact, e.g., to take our famous caption example, Axel could provide the very first version of his package as caption-⟨*some-date*⟩.sty and caption2 as caption-⟨*another-date*⟩.sty in addition to adding the necessary declarations to the current release.

The necessary declarations in the main file are provided by the two commands, \DeclareRelease, and \DeclareCurrentRelease, that must be used in a *release selection* section at the beginning of the file. For each old release you can specify a ⟨*name*⟩, the ⟨*date*⟩ when it was first available and the ⟨*external-file*⟩ that contains the code.

[3] Alternatively you could try to roll the installation forward, by using a current latexrelease package together with a suitable date option.

[4] Instead of making a verbatim copy you may want to adjust the commentary added by docstrip at the top of the file. Though technically correct, it is a bit misleading if the file still contains the phrase "was generated from ...", given that it is now a frozen version representing a particular state in time, rather than being a generated one that can be regenerated any time as necessary.

```
\DeclareRelease
      {⟨name⟩}{⟨date⟩}{⟨external-file⟩}
```

Either the ⟨*name*⟩ or the ⟨*date*⟩ can be empty, but not both at the same time. Not specifying a ⟨*date*⟩ is mainly intended for providing "beta" versions that people can explicitly select but that should play no role in date rollbacks.

The current release also gets a declaration, but this time with only two arguments: a ⟨*name*⟩ (again possibly empty) and a ⟨*date*⟩ since the code for this release will be the rest of the current file:

```
\DeclareCurrentRelease{⟨name⟩}{⟨date⟩}
```

This declaration has to be the last one in sequence as it will end the *release selection* processing.

The order of the other releases has to be from the oldest to the newest since the loading mechanism compares every release declaration with the target rollback date and stops the moment it finds one that is newer than this target date. It will then select the one before, i.e., the last one that is at least as old as the target. Since the \DeclareRelease declarations with an empty ⟨*date*⟩ argument do not play a role in date rollbacks, they can be placed anywhere within the sequence.

As a typical example of a release section the start of the multicol package currently looks as follows because there was a major internal rewrite in April 2018. Note that because of some minor fixes afterwards the actual package date is already June.

```
\NeedsTeXFormat{LaTeX2e}[2018-04-01]
\DeclareRelease{}{2017-04-11}
                 {multicol-2017-04-11.sty}
\DeclareCurrentRelease{}{2018-04-01}
\ProvidesPackage{multicol}[2018/06/26 v1.8t
             multicolumn formatting (FMi)]
```

If the rollback target is not a date but a name, the mechanism works in the same way with the exception that a release is selected only if the name matches. If none of the names is a match, then the mechanism will raise an error and continue by using the current release.

5 Special considerations for developers

While loading an older release of a package or class, both types of release declarations are made no-ops, so that, in case the files containing the code also have such declarations, they will not be looked at or acted upon. This makes it possible to simply move the code from an old release into a new file without the need to touch it at all. Of course, removing those declarations doesn't hurt and will make loading a tiny fraction faster.

As mentioned earlier, best practice for release names is to append the release date to the package or class name, but the ⟨*external file*⟩ argument also allows other naming schemes.

You may have wondered why you have to make a declaration for the current release, given that later on there will be a `\Provides...` declaration that also contains a date and a version string and thus could signal the end of the release declaration section. The reason is as follows: if you want to give your current release a name, then it is best practice to make that name something simple like `v4` (and keep it that way) even though your current package is technically already at `v4.2c` and is listed that way in the `\ProvidesPackage` declaration. For the same reason (given that not every minor change will be provided as a separate version to which people can roll back), the ⟨*date*⟩ in `\DeclareCurrentRelease` reflects when that major release was first introduced. Thus, after a while that date may well be earlier than the current package date.

5.1 Early adopters

For one or two years after the introduction of this new method, there is a danger that people with older installations will pick up an individual package from, say, CTAN that contains release declarations with which their kernel (from 2017 or earlier) is unable to cope. It may therefore be a good idea for developers to additionally add the following lines at the top of packages or classes when using the new rollback feature:

```
\providecommand\DeclareRelease[3]{}
\providecommand\DeclareCurrentRelease[2]{}
```

This way the declarations will be bypassed in case the kernel doesn't know how to deal with them.

As an alternative one could add a statement that requires a minimal kernel version, i.e.:

```
\NeedsTeXFormat{LaTeX2e}[2018-04-01]
```

so that users get a clear error message that they need to update their installation if they want to use the current file.

5.2 New major release in beta

If you are working on a new major release of your package or class, you may want to get it out into the open so that people can try it and provide feedback. In that case the current release is still the official release which should be selected by default, and the "beta" version should only be selected if explicitly requested. To achieve that you could add

`\DeclareRelease{beta}{}{`⟨*external-file*⟩`}`

before

`\DeclareCurrentRelease{}{`⟨*some-date*⟩`}`

so that testers can explicitly access your new version by asking for it via

`\usepackage[`⟨*options*⟩`]{`⟨*package*⟩`}[=beta]`

while everyone loading the package without the extra optional argument would get the current release.

5.3 Two major releases in use

One special scenario for which this method is only partially suitable is the case where we have two major releases that are in continuing parallel use and that are both under active maintenance (i.e., receive bug fixes and other updates once in a while). In that case it is necessary to make one version the primary release and allow the other (and its updates) to be accessed only via names: a date rollback can obviously work for only one line of development.

For example, if both `v4` and `v5` of package foo are in use and you consider `v5` as being the go-forward version (even though you are still fixing bugs in the `v4` code), then you can deploy a strategy as in the following example:

```
% last v4 only release:
 \DeclareRelease{}{2017-06-23}
                  {foo-2017-06-23.sty}
% first v5 release:
 \DeclareRelease{}{2017-08-01}
                  {foo-2017-08-01.sty}
% patch to v4 after v5 got introduced:
 \DeclareRelease{v4.1}{}
                  {foo-v4-2017-09-20.sty}
% patch to v5:
 \DeclareRelease{}{2017-08-25}
                  {foo-2017-08-25.sty}
% another patch to v4:
 \DeclareRelease{v4.2}{}
                   {foo-v4-2017-10-01.sty}
% nickname for the latest v4 if you want
% users to have simple access via a name:
 \DeclareRelease{v4}{}
                  {foo-v4-2017-10-01.sty}
% current v5 with further patches:
 \DeclareCurrentRelease{v5}{2018-01-01}
```

This way users can use `\usepackage{foo}[=v4]` to get the latest v4 release or use the more detailed release names such as `[v4.1]`. This means that if package foo is requested at version v4 (or one of its sub-releases), it will not change even if there is a general rollback request via latexrelease.

Normally, this should be just fine, but if you really require automatic date rollback functionality on both major versions, because the two are really equal in rank, then you are essentially saying they are independent works with some common root. In that case you should give them two separate names,

e.g., call the older version foo-v4 when you introduce version 5 of foo and from that point on manage the history independently.[5]

5.4 Fine grained control (if needed)

As mentioned earlier, the interface is deliberately designed to be simple and easy to use. As a price, each rollback point is (by default) a separate file. The idea behind this is that there is not much point in managing each and every small change as a rollback point, but only those that possibly alter the behavior of a package within the document so that, when processing older documents, it is important to be able to get back to an earlier state.

However, if you find yourself in a situation where you have many rollback files with only minor differences, and you consider this unsatisfactory, then here is one other command at your disposal that you can use to combine several files into a single file. Within a file corresponding to a `\DeclareRelease` declaration you can use

```
\IfTargetDateBefore{⟨date⟩}
    {⟨before-date-code⟩}{⟨after-or-at-date-code⟩}
```

This must be used after the *release selection* section (if present) and has the following effect: If the user requested, say, `[=2017-06-01]` then the mechanism first selects the file that is supposed to be current on that date, i.e., the release that was introduced on that date or is the last one that was introduced before that date. Now, if in this file we have a statement like the above, then the ⟨*date*⟩ is compared to `2017-06-01` and depending on the outcome either ⟨*before-date-code*⟩ or ⟨*after-or-at-date-code*⟩ is executed.

This way a single external file can hold rollback information for several patches on distinct dates, but of course, the burden is then on the developer to add the appropriate declarations, which is a little more work than just copying and renaming files.

The alternative is to use `\IncludeInRelease` and `\EndIncludeInRelease`. The latexrelease package documentation [1] gives some advice on how to apply those commands.

5.5 Using `l3build` for source management

If you use `l3build` [2] for managing your sources, then it is necessary to ensure that the files for the old releases are copied into the distribution. To support this, the default configuration for `l3build` specifies

```
sourcefiles = {"*.dtx", ".ins",
                 "*-????-??-??.sty"}
```

i.e., all `.dtx` and `.ins` files, together with all `.sty` files matching the naming convention suggested in this article, are automatically included in the build.

If you prefer a different naming convention you have to adjust this setting in the `build.lua` file of your project. Otherwise you are ready to go without any adjustments.

6 Command summary

6.1 Document interface, for users

For a global rollback of kernel and packages, use

```
\RequirePackage[⟨target-date⟩]{latexrelease}
```

at the beginning of your document.

To request a rollback for a single package or class, use the second optional argument with the date preceded by an equal sign, i.e.,

```
\documentclass[⟨options⟩]{⟨class⟩}[=⟨date⟩]
\usepackage [⟨options⟩]{⟨package⟩}[=⟨date⟩]
```

6.2 Package and class interface, for developers

To declare an old or special release, use

```
\DeclareRelease
        {⟨name⟩}{⟨date⟩}{⟨external-file⟩}
```

Leave the ⟨*name*⟩ argument empty if rollback should be only via dates. Leave the ⟨*date*⟩ empty if this special release should be accessible only via its name.

Always finish this *release selection* section with a declaration for the current release:

```
\DeclareCurrentRelease{⟨name⟩}{⟨date⟩}
```

In this declaration you must provide a ⟨*date*⟩ but the ⟨*name*⟩ can be left empty (which is the usual case).

Within a release file (but after the *release selection* section), you can specify conditional code to be selected based on a requested rollback date by using:

```
\IfTargetDateBefore{⟨date⟩}
    {⟨before-date-code⟩}{⟨after-or-at-date-code⟩}
```

References

[1] The LaTeX Team. *The* `latexrelease` *package*, April 2015. Available at `https://www.latex-project.org/help/documentation`.

[2] The LaTeX Team. *The* `l3build` *package — Checking and building packages*, March 2018. The manual `l3build.pdf` should be part of your installation. Run "`texdoc l3build`" to find it. See also `https://ctan.org/pkg/l3build`.

◇ Frank Mittelbach
`https://www.latex-project.org`

[5] While in rare cases this might be the best approach, try to avoid it as long term management will be problematic, to say the least.