verdepcheck

Pawel Rucki

July 20, 2023

Agenda

1. Motivation
2. Solution
3. How to use?
4. How it works?
5. Discussion
6. Future work

Motivation

```
Type: Package
Package: foo.package
Version: 1.2.3.9000
Depends:
    R (>= 3.6)
Imports:
    dplyr,
    (...)
```

The above specification indicates compatibility with **all** versions of `dplyr` which is very unlikely to be true.

Motivation

```
Error: object ‘foo’ is not exported by 'namespace:bar'
```

```
`foo()` was deprecated in bar 1.0.0.
i Please use `baz()` instead.
```

The new versions of dependencies might introduce a breaking changes.

Motivation

Many interdependent packages actively developed in parallel - when to increase dependency version requirement?
Blindly increase all requirements might be incorrect as one package might be fully compatible with the older version of the other one.
Keeping all unchanged is incorrect as there might be some *breaking* changes.

Motivation

R environments for clinical trials are usually not updated very frequently.
Users might use `renv` for environment management and packages used might not get updates for some time.
As a result, your package might be used with *older* version of dependent packages.

Solution

`verdepcheck` (**ver**sion of **dep**endencies **check**) - a tool for package developers to check your package against different set of dependencies. It is meant to be used primarily in CI.
Available at:

- **`insightsengineering/verdepcheck`**
- **`insightsengineering/r-verdepcheck-action`**

Important

Do not confuse `verdepcheck` with `revdepcheck` for reverse dependency checks. Those are totally different tools!

What it is not about:

- This does not perform matrix testing of *all* versions of dependent packages.
- This only looks into package versions - in particular: it does not check R versions or system architectures.

Let's call a "different set of dependencies" a *strategy*. Currently there are three *strategies* implemented:

- **max** - use *development* version of dependencies
- **release** - use *released* version of dependencies (e.g. from CRAN)
- **min** - use *minimal* version of dependencies as per `DESCRIPTION` file

This tool is looking into a **direct** dependencies only.

What are the benefits?

- It forces you to specify minimal version of package dependencies to prevent incorrect package usage.
- Incorporate information about upcoming breaking changes.

What's needed?

For **max** strategy the algorithm will look for dependent packages references in a new section `Config/Needs/verdepcheck` of the `DESCRIPTION` file.
For **release** and **min** it's not needed (at least for CRAN packages).

Note

This might indicate some duplication with the `Remotes` section but it cannot be used because of side effects, i.e. configuration of `verdepcheck` should have no impact package installation.

How to use?

`<strategy>_deps_check(path)`

```
library(verdepcheck)

max_deps_check("/path/to/package")
release_deps_check("/path/to/package")
min_deps_check("/path/to/package")
```

How to use?

Multiple steps (note private function calls):

```
library(verdepcheck)

x <- new_max_deps_installation_proposal("/path/to/package")

# resolve dependencies
verdepcheck:::solve_ip(x)

# download dependencies
verdepcheck:::download_ip(x)

# resolve and install
verdepcheck:::install_ip(x)

# run R CMD CHECK
verdepcheck:::check_ip(x)
```

How to use the output?

The main `<strategy>_deps_check` function returns a list of two elements:

- `"ip"` - installation proposal object from `pkgdepends` - see **docs**
- `"check"` - returned value from `rcmdcheck::rcmdcheck()` - see **docs**

```
x <- max_deps_check("/path/to/package")

# show dependency resolution
x$ip$show_solution()
x$ip$draw()

# create artifact
x$ip$create_lockfile("/path/to/pkg.lock")

# print R CMD CHECK results
x$check$session_info
x$check$status
```

How it works?

This package is heavily based on **`pkgdepends`** for dependency detection and resolution and also **`rcmdcheck`** for executing `"R CMD CHECK"`.
It also uses other packages like `pkgcache`, `pkgbuild`, `desc` or `cli`.

Please also see: **r-lib/rcmdcheck/issues/195** and **r-lib/pkgdepends/issues/305**.
Workarounds have been implemented.

The algorithm

1. Read package dependencies (alongside minimal versions) from `DESCRIPTION` file using Depends, Imports, Suggests sections.
2. Derive package reference according to the strategy used.
3. Resolve dependency tree.
4. Download package sources.
5. Build and install into temporary directory.
6. Execute `"R CMD CHECK"` using directory from the previous step as a library path.

Package reference format

We are using `pkgdepends` (which is then calling `pak`) for package installation. It uses specific format for **package references**.
Few examples:

```
dplyr
dplyr@1.0.0
cran::dplyr

foo/bar
foo/bar@12ab3456
github::foo/bar

bioc::S4Vectors

/absolute/path/to/package/dir
local::.

deps::.
```

Find package references - the algorithm

- **max** (`verdepcheck:::get_ref_max()`)
  Use reference provided in `Config/Needs/verdepcheck` as is.
- **release** (`verdepcheck:::get_ref_release()`)
  For CRAN pkg: use standard reference (i.e. package name only).
  For GitHub pkg: find the latest release reference.
- **min** (`verdepcheck:::get_ref_min()`)
  For CRAN pkg: use the lowest possible from CRAN archive for which version condition is met.
  For GitHub pkg: loop through releases (tags if no releases) starting from the oldest one until version condition is met.
  Please note that only this strategy includes search on versions of dependencies.

Few examples

Imports:	Config	ref max	ref release	ref min
`rlang (>= 1.0.0)`	`r-lib/rlang`	`r-lib/rlang`	`rlang`	`rlang@1.0.0`
`rlang`	`r-lib/rlang`	`r-lib/rlang`	`rlang`	`rlang@0.1.0`
`rlang (>= 1.0.0)`		`rlang`	`rlang`	`rlang@1.0.0`
`rlang`		`rlang`	`rlang`	`rlang@0.1.0`
`foo`	`bar/foo`	`bar/foo`	`bar/foo@v1.2.3`	`bar/foo@v0.1.0`
`foo`		ERROR: package not found	ERROR: package not found	ERROR: package not found

Imports:	Config	ref max	ref release	ref min
`Foo`	`bioc::Foo`	`bioc::Foo`	`bioc::Foo`	`bioc::Foo`

Dependency resolution

Dependency resolution will *collapse* the whole dependency tree (including indirect dependencies) and warn about conflicts.

Important

You have to specify minimal versions allowing dependency resolution to complete.

It could be a process of multiple iterations of increasing dependent package versions.

Explanation for direct dependencies

The initial prototype of `min` strategy included recursive find of minimal dependency.
This oftentimes lead into usage of already archived package or not able to compile very old package on modern systems architecture.
In order to solve potential issues, package author would need to control / specify indirect dependencies which is counter intuitive.
Therefore only **direct** dependencies are being looked into and **indirect** dependencies are installed using standard way - i.e. the greatest available.

Explanation for direct dependencies

It might happen that the environment would include relatively old version of a directly dependent package alongside the newest version of indirect dependency.
E.g.

```
Type: Package
Package: foo.package
Version: 1.2.3.9000
Depends:
    R (>= 3.6)
Suggests:
    testthat (>= 2.0.0)
    (...)
```

As a result we will use `testthat@2.0.0` (direct dependency) released in 2017 alongside the newest release of `rlang` (indirect dependency - required by `testthat`) from 2023.

Explanation for direct dependencies

It’s definitely arguable how likely that scenario is but it’s definitely a valid and fully functional environment in which your package might be used.
There is an idea for a new `min_concurrent` strategy to address that - not discussed here.

Example - failure

```
Type: Package
Package: foo.package
Version: 1.2.3.9000
Depends:
    R (>= 3.6)
Imports:
    dplyr (>= 1.0.0),
    tibble (>= 1.0.0)
    (...)
```

```
Type: Package
Package: dplyr
Package: 1.0.0
Depends:
    R (>= 3.2.0)
Imports:
    (...)
    tibble (>= 2.1.3)
    (...)
```

```r
x <- pkgdepends::new_pkg_deps(c("dplyr@1.0.0", "tibble@1.0.0"))
x$solve()
x$get_solution()
```

```
<pkg_solution>
+ result: FAILED
+ refs:
  - dplyr@1.0.0
  - tibble@1.0.0
+ constraints (93):
  (...)
x failures:
* dplyr@1.0.0: Can't install dependency tibble (>= 2.1.3)
* tibble@1.0.0: Conflicts with tibble@1.0.0
* tibble: Conflicts with tibble@1.0.0
```

Example - success

```
Type: Package
Package: foo.package
Version: 1.2.3.9000
Depends:
    R (>= 3.6)
Imports:
    dplyr (>= 1.0.0),
    tibble (>= 2.1.3)
    (...)
```

```
Type: Package
Package: dplyr
Package: 1.0.0
Depends:
    R (>= 3.2.0)
Imports:
    (...)
    tibble (>= 2.1.3)
    (...)
```

```
x <- pkgdepends::new_pkg_deps(c("dplyr@1.0.0", "tibble@2.1.3"))
x$solve()
x$get_solution()
```

```
<pkg_solution>
+ result: OK
+ refs:
  - dplyr@1.0.0
  - tibble@2.1.3
+ constraints (95):
  - select dplyr exactly once
  - select tibble exactly once
  - select R6 at most once
  - select cli at most once
  - select ellipsis at most once
  - select fansi at most once
  - select generics at most once
  - select glue at most once
  - select lifecycle at most once
  - select magrittr at most once
  ...
+ solution:
  - cli
```

Discussion on the min strategy

- One might argue that `foo.package` is fully compatible with the `tibble@1.0.0` syntax but `tibble@1.0.0` cannot be installed alongside `dplyr@1.0.0` hence `foo.package` cannot be used with both `dplyr@1.0.0` and `tibble@1.0.0` loaded.
  Therefore `tibble (>= 1.0.0)` (alongside `dplyr (>= 1.0.0)`) really indicates `tibble (>= 2.1.3)`.
- *Collapsing* the whole dependency tree requires you to specify `tibble (>= 2.1.3)` otherwise the dependencies cannot be resolved successfully.

Discussion on the min strategy

All of this makes the minimal version specification of dependencies truly a **installable** and **compatible** set of dependencies rather than **compatibility**-wise set od dependencies.
It is a little more strict (i.e. includes additional version increases) as well as breaks encapsulation of what is specified where.

Important

This is a significant limitation of the current implementation.

Real-life observations:

- Oftentimes the dependency tree is of a big depth and there are multi-step conflicts.
- Some very old packages are not compilable on modern machines and you have to increase versions to make it installable.
- The package resolution is just the first step. The next one is to use them in `R CMD CHECK` and validate its compatibility.
- In order to enter GHA debug lines, `verdepcheck` will continue on error entering `R CMD CHECK` step when e.g. dependency resolve failed. If you encounter `vignette builder 'knitr' not found` error - most likely you failed on resolve or install.

A few caveats:

None of the test result is *stable*:

- **min**
  Indirect dependencies are installed as usual using the newest available. Therefore: it is not guaranteed that the new release of indirect dependency will be fully compatible with some old version of direct dependency.
  e.g. future version of `tibble` won't be fully compatible with `dplyr@1.0.0`
- **release**
  The same as above though it's relatively unlikely if you use CRAN-only dependencies as this is checked by CRAN.
- **max**
  This is the most instable environment changed by pushes to any of the dependencies. Also: the main branch is usually not guaranteed to be correct and you might pull broken package version.

  Therefore, this test should be triggered periodically and not on code change (unless changes includes dependencies). Triggering it on change might give a false message that a failure stems from code changes.

Future work:

- Idea: add `branch` strategy with recursive resolve for possible replacement of `staged.dependencies`.
- Idea: `concurrent_min` strategy that uses versions of recursive dependencies that were concurrent (as opposed to max) with the versions of the direct deps being tested.
- Idea: add custom strategies.
- Enhance `r-verdepcheck-action` output readability.
- Remove workarounds implemented once referenced issues are closed.

THANK YOU!

Error