Package: verdepcheck (via r-universe)

September 16, 2024

Type Package

Title Check Package using Various Versions of Dependencies

Version 0.0.0.9001

Date 2023-02-14

Description Derive package dependencies from DESCRIPTION file using various strategies and run ``R CMD CHECK'' to validate package compatibility.

License Apache License 2.0 | file LICENSE

URL https://github.com/insightsengineering/verdepcheck/

BugReports https://github.com/insightsengineering/verdepcheck/issues

Depends R (>= 3.6)

Imports cli (>= 3.6.0), desc (>= 1.2), gh, jsonlite, pkgcache (>= 2.2.2.9000), pkgdepends (>= 0.5.0), rcmdcheck, remotes (>= 2.2.0), stats, tools, utils, withr (>= 2.4.3)

Suggests knitr (>= 1.42), pingr, rmarkdown (>= 2.23), testthat (>= 3.0.4)

Remotes r-lib/pkgcache

Config/Needs/verdepcheck r-lib/cli, r-lib/desc, r-lib/gh, jeroen/jsonlite, r-lib/pkgcache, r-lib/pkgdepends, r-lib/rcmdcheck, r-lib/remotes, r-lib/withr, yihui/knitr, r-lib/pingr, r-lib/testthat

VignetteBuilder knitr

Encoding UTF-8

Language en-US

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.2

Config/testthat/edition 3

Repository https://insightsengineering.r-universe.dev

RemoteUrl https://github.com/insightsengineering/verdepcheck

RemoteRef main

RemoteSha 1f072414f4df9b26f9abca0200770e98fc3a1d47

Contents

`check_ip` *Executes* `rcmdcheck::rcmdcheck()` *on a local package using* `libpath` *from the installation plan.*

Description

Executes `rcmdcheck::rcmdcheck()` on a local package using `libpath` from the installation plan.

Usage

```
check_ip(ip, path, build_args = character(), check_args = character(), ...)
```

Arguments

`ip`	(`pkg_installation_plan`) object to execute
`path`	(`string`) path to the package sources
`build_args`	(`string`) value passed as `build_args` argument into `rcmdcheck::rcmdcheck()`
`check_args`	(`string`) value passed as `args` argument into `rcmdcheck::rcmdcheck()`
`...`	other arguments passed to `rcmdcheck::rcmdcheck()`

Value

An S3 object (list) with fields `errors`, `warnings` and `notes`. These are all character vectors containing the output for the failed check.

See Also

`rcmdcheck::rcmdcheck()` for other configuration possibilities

`download_ip` *Solve installation plan ignoring entries with "@*release" remote refs for detected conflicts.*

Description

Solve installation plan ignoring entries with "@*release" remote refs for detected conflicts.

Usage

```
download_ip(ip)
```

Arguments

`ip`	(`pkg_installation_plan`) object to execute

Value

`pkg_installation_plan` object invisibly

`execute_ip` *Executes installation plan and `rcmdcheck::rcmdcheck()` in "try mode" to always return.*

Description

Executes installation plan and `rcmdcheck::rcmdcheck()` in "try mode" to always return.

Usage

```
execute_ip(ip, path, build_args, check_args, ...)
```

Arguments

`ip`	(`pkg_installation_plan`) object to execute
`path`	(`string`) path to the package sources
`build_args`	(`string`) value passed as `build_args` argument into `rcmdcheck::rcmdcheck()`
`check_args`	(`string`) value passed as `args` argument into `rcmdcheck::rcmdcheck()`
`...`	other arguments passed to `rcmdcheck::rcmdcheck()`

Value

a named `list` with two elements:

- `"ip"` - installation plan object
- `"check"` - returned value from `rcmdcheck::rcmdcheck()`

`get_avail_date` *Get available date for the package.*

Description

Oftentimes, the release date of the package does not correspond to the date when the package is available in the PPM. Usually it takes one day for the PPM to sync with the CRAN. This function will return the date when the package is available in the PPM.

Usage

```
get_avail_date(remote_ref, start = get_release_date(remote_ref))

## S3 method for class 'remote_ref_cran'
get_avail_date(remote_ref, start = get_release_date(remote_ref))

## S3 method for class 'remote_ref_standard'
get_avail_date(remote_ref, start = get_release_date(remote_ref))

## S3 method for class 'remote_ref'
get_avail_date(remote_ref, start = get_release_date(remote_ref))
```

Arguments

`remote_ref`	(`remote_ref`) object created with `pkgdepends::parse_pkg_ref()`
`start`	(`Date`) optional, the date when the package was released

Value

Date. This can be safely used as a snapshot date for the PPM.

Examples

```
get_avail_date(pkgdepends::parse_pkg_ref("cran::dplyr"))
get_avail_date(pkgdepends::parse_pkg_ref("cran::dplyr@1.1.0"))


get_avail_date(pkgdepends::parse_pkg_ref("dplyr"))
get_avail_date(pkgdepends::parse_pkg_ref("dplyr@1.1.0"))
get_avail_date(pkgdepends::parse_pkg_ref("bioc::MultiAssayExperiment"))
get_avail_date(pkgdepends::parse_pkg_ref("tidyverse/dplyr@v1.1.0"))
```

`get_ref_max` *Get reference to the maximal version of the package.*

Description

Get reference to the maximal version of the package.

Usage

```
get_ref_max(remote_ref)
```

Arguments

`remote_ref`	(`remote_ref`) object created with `pkgdepends::parse_pkg_ref()`

Value

(`remote_ref`) object with the package reference

Examples

```
get_ref_max(pkgdepends::parse_pkg_ref("dplyr"))
get_ref_max(pkgdepends::parse_pkg_ref("cran::dplyr"))
get_ref_max(pkgdepends::parse_pkg_ref("tidyverse/dplyr"))
get_ref_max(pkgdepends::parse_pkg_ref("bioc::MultiAssayExperiment"))
```

`get_ref_min` *Get reference to the minimal version of the package.*

Description

Get reference to the minimal version of the package.

- for standard and CRAN-type of remote - this would use `pkgcache::cran_archive_list()` to obtain historical data.
- for GitHub type of remote - this would use `gh::gh_gql()` to get list of all releases or tags and then `gh::gh()` to download `DESCRIPTION` file and then read package version.

Usage

```
get_ref_min(remote_ref, op = "", op_ver = "")

## S3 method for class 'remote_ref'
get_ref_min(remote_ref, op = "", op_ver = "")

## S3 method for class 'remote_ref_cran'
get_ref_min(remote_ref, op = "", op_ver = "")

## S3 method for class 'remote_ref_standard'
get_ref_min(remote_ref, op = "", op_ver = "")

## S3 method for class 'remote_ref_github'
get_ref_min(remote_ref, op = "", op_ver = "")
```

Arguments

`remote_ref`	(`remote_ref`) object created with `pkgdepends::parse_pkg_ref()`
`op`	(`character(1)`) optional, version condition comparison operator (e.g. ">", ">=")
`op_ver`	(`character(1)`) optional, version number against which op argument is applied

Value

(`remote_ref`) object with the package reference

See Also

`get_ref_min_incl_cran()`

Examples

```
get_ref_min(pkgdepends::parse_pkg_ref("bioc::MultiAssayExperiment"))

get_ref_min(pkgdepends::parse_pkg_ref("cran::dplyr"))


get_ref_min(pkgdepends::parse_pkg_ref("dplyr"))


get_ref_min(pkgdepends::parse_pkg_ref("tidyverse/dplyr"))
```

get_ref_min_incl_cran *Get reference to the minimal version of the package including also check in CRAN repository.*

Description

Get reference to the minimal version of the package including also check in CRAN repository.

Usage

```
get_ref_min_incl_cran(remote_ref, op = "", op_ver = "")

## S3 method for class 'remote_ref'
get_ref_min_incl_cran(remote_ref, op = "", op_ver = "")

## S3 method for class 'remote_ref_github'
get_ref_min_incl_cran(remote_ref, op = "", op_ver = "")
```

Arguments

remote_ref	(remote_ref) object created with pkgdepends::parse_pkg_ref()
op	(character(1)) optional, version condition comparison operator (e.g. ">", ">=")
op_ver	(character(1)) optional, version number against which op argument is applied

Value

(remote_ref) object with the package reference

See Also

get_ref_min_incl_cran()

Examples

```
verdepcheck:::get_ref_min_incl_cran(pkgdepends::parse_pkg_ref("dplyr"))

verdepcheck:::get_ref_min_incl_cran(pkgdepends::parse_pkg_ref("bioc::MultiAssayExperiment"))

verdepcheck:::get_ref_min_incl_cran(pkgdepends::parse_pkg_ref("tidyverse/dplyr"))
```

get_ref_release — *Get reference to the release version of the package.*

Description

Get reference to the release version of the package.

Usage

```
get_ref_release(remote_ref)
```

Arguments

`remote_ref`	(`remote_ref`) object created with `pkgdepends::parse_pkg_ref()`

Value

(`remote_ref`) object with the package reference

Examples

```
get_ref_release(pkgdepends::parse_pkg_ref("dplyr"))
get_ref_release(pkgdepends::parse_pkg_ref("cran::dplyr"))
get_ref_release(pkgdepends::parse_pkg_ref("tidyverse/dplyr"))
get_ref_release(pkgdepends::parse_pkg_ref("bioc::MultiAssayExperiment"))
```

get_release_date — *Get release date.*

Description

Get release date.

Usage

```
get_release_date(remote_ref)

## S3 method for class 'remote_ref_github'
get_release_date(remote_ref)

## S3 method for class 'remote_ref_cran'
get_release_date(remote_ref)

## S3 method for class 'remote_ref_standard'
get_release_date(remote_ref)
```

```
## S3 method for class 'remote_ref_bioc'
get_release_date(remote_ref)

## S3 method for class 'remote_ref'
get_release_date(remote_ref)
```

Arguments

`remote_ref`	(`remote_ref`) object created with `pkgdepends::parse_pkg_ref()`

Value

Date

Examples

```
get_release_date(pkgdepends::parse_pkg_ref("tidyverse/dplyr@v1.1.0"))


get_release_date(pkgdepends::parse_pkg_ref("cran::dplyr"))
get_release_date(pkgdepends::parse_pkg_ref("cran::dplyr@1.1.0"))


get_release_date(pkgdepends::parse_pkg_ref("dplyr"))


get_release_date(pkgdepends::parse_pkg_ref("MultiAssayExperiment"))
```

`get_version` *Get package version.*

Description

Get package version.

Usage

```
get_version(remote_ref)

## S3 method for class 'remote_ref'
get_version(remote_ref)
```

Arguments

`remote_ref`	(`remote_ref`) object created with `pkgdepends::parse_pkg_ref()`

Value

Package version created with `package_version`.

Examples

```
get_version(pkgdepends::parse_pkg_ref("dplyr"))
get_version(pkgdepends::parse_pkg_ref("tidyverse/dplyr"))
get_version(pkgdepends::parse_pkg_ref("tidyverse/dplyr@v1.1.0"))
get_version(pkgdepends::parse_pkg_ref("bioc::MultiAssayExperiment"))
```

`install_ip` *Executes installation plan.*

Description

This function would executes the following:

- solves package dependencies
- downloads all package dependencies
- installs system requirements
- installs all package dependencies

Usage

```
install_ip(ip)
```

Arguments

`ip`	(`pkg_installation_plan`) object to execute

Value

`pkg_installation_plan` object invisibly

`max_deps_check` *Execute `R CMD CHECK` on a local package with all dependencies pre-installed using various strategies.*

Description

Execute `R CMD CHECK` on a local package with all dependencies pre-installed using various strategies.

Usage

```
max_deps_check(
  path,
  extra_deps = character(),
  config = list(),
  build_args = character(),
  check_args = character(),
  ...
)

release_deps_check(
  path,
  extra_deps = character(),
  config = list(),
  build_args = character(),
  check_args = character(),
  ...
)

min_cohort_deps_check(
  path,
  extra_deps = character(),
  config = list(),
  build_args = character(),
  check_args = character(),
  ...
)

min_isolated_deps_check(
  path,
  extra_deps = character(),
  config = list(),
  build_args = character(),
  check_args = character(),
  ...
)
```

Arguments

`path`	(`string`) path to the package sources
`extra_deps`	(`character(1)`) Extra dependencies specified similarly as in the `DESCRIPTION` file, i.e. `"<package name> (<operator> <version>)"` where both `<operator>` and `<version>` are optional. Multiple entries are possible separated by `";"`.
`config`	(`list`) configuration options. See `pkgdepends::pkg_config` for details. `"dependencies"` and `"library"` elements are overwritten by package level defaults.
`build_args`	(`string`) value passed as `build_args` argument into `rcmdcheck::rcmdcheck()`
`check_args`	(`string`) value passed as `args` argument into `rcmdcheck::rcmdcheck()`
`...`	other arguments passed to `rcmdcheck::rcmdcheck()`

Value

a named `list` with two elements:

- `"ip"` - installation plan object
- `"check"` - returned value from `rcmdcheck::rcmdcheck()`

strategies

Currently implemented strategies:

- `max` - use the greatest version of dependent packages. Please note that using development version is not guaranteed to be stable. See get_ref_max for details.
- `release` - use the released version of dependent packages. It will try use CRAN if possible else if GitHub release is available then use it else fail. See get_ref_release for details.
- `min_cohort` - find maximum date of directly dependent packages release dates and use that as PPM snapshot date for dependency resolve.
- `min_isolated` - for each direct dependency: find its release date and use it as PPM snapshot for resolving itself. Next, combine all the individual resolutions and resolve it altogether again.

Both "min" strategies relies on PPM snapshot in order to limit the versions of indirect dependencies so that dependency resolution ends with a package released no earlier than any of its dependency. However, that's not always true for `min_isolated` strategy - done on purpose.

Please note that only `min_cohort` and `min_isolated` strategies are "stable". The rest are basing on dynamic references therefore it results might be different without changes in tested package. The most straightforward example is `max` strategy in which the environment will be different after any push of any of the dependencies.

See Also

deps_installation_proposal

Examples

```
x <- max_deps_check(".")
x$ip
x$check


x <- release_deps_check(".")
x$ip
x$check


x <- min_cohort_deps_check(".")
x$ip
x$check


x <- min_isolated_deps_check(".")
x$ip
x$check
```

`new_max_deps_installation_proposal`

Create installation proposal using various dependency strategies

Description

These functionalities would read local package `DESCRIPTION` file, derive dependencies from `"Config/Needs/verdepcheck"` and create an installation proposal using various strategies for package versions as described below.

Usage

```
new_max_deps_installation_proposal(
  path,
  extra_deps = character(0L),
  config = list()
)

new_release_deps_installation_proposal(
  path,
  extra_deps = character(0L),
  config = list()
)

new_min_cohort_deps_installation_proposal(
  path,
  extra_deps = character(0L),
  config = list()
)

new_min_isolated_deps_installation_proposal(
  path,
  extra_deps = character(0L),
  config = list()
)
```

Arguments

`path`	(`string`) path to the package sources
`extra_deps`	(`character(1)`) Extra dependencies specified similarly as in the `DESCRIPTION` file, i.e. `"<package name> (<operator> <version>)"` where both `<operator>` and `<version>` are optional. Multiple entries are possible separated by `";"`.
`config`	(`list`) configuration options. See `pkgdepends::pkg_config` for details. `"dependencies"` and `"library"` elements are overwritten by package level defaults.

Value

`pkg_installation_plan` object

strategies

Currently implemented strategies:

- `max` - use the greatest version of dependent packages. Please note that using development version is not guaranteed to be stable. See get_ref_max for details.
- `release` - use the released version of dependent packages. It will try use CRAN if possible else if GitHub release is available then use it else fail. See get_ref_release for details.
- `min_cohort` - find maximum date of directly dependent packages release dates and use that as PPM snapshot date for dependency resolve.
- `min_isolated` - for each direct dependency: find its release date and use it as PPM snapshot for resolving itself. Next, combine all the individual resolutions and resolve it altogether again.

Both "min" strategies relies on PPM snapshot in order to limit the versions of indirect dependencies so that dependency resolution ends with a package released no earlier than any of its dependency. However, that's not always true for `min_isolated` strategy - done on purpose.

Please note that only `min_cohort` and `min_isolated` strategies are "stable". The rest are basing on dynamic references therefore it results might be different without changes in tested package. The most straightforward example is `max` strategy in which the environment will be different after any push of any of the dependencies.

configuration

`verdepcheck` will look into `"Config/Needs/verdepcheck"` field of the `DESCRIPTION` file for dependent packages references. See `pkgdepends::pkg_refs` for details and this package `DESCRIPTION` file for an example. Please note that some features are enabled only for package references from GitHub. If you specify additional details (i.e. tag, commit, PR or `@*release`) in the reference then it wouldn't be changed. Therefore, in order to make full use of various strategies, it is recommended to specify general reference in form of `[<package>=][github::]<username>/<repository>[/<subdir>]` - i.e. without `[<detail>]` part. Please see also `pkgdepends::pkg_config` and `pak::pak-config` for other configuration possibilities.

See Also

pkgdepends::pkg_installation_proposal

Examples

```
x <- new_max_deps_installation_proposal(".")
x$solve()
x$get_solution()


x <- new_release_deps_installation_proposal(".")
x$solve()
x$get_solution()


x <- new_min_cohort_deps_installation_proposal(".")
solve_ip(x)
x$get_solution()


x <- new_min_isolated_deps_installation_proposal(".")
solve_ip(x)
x$get_solution()
```

`solve_ip` *Try to solve using standard method. If error - use resolve_ignoring_release_remote.*

Description

Try to solve using standard method. If error - use resolve_ignoring_release_remote.

Usage

```
solve_ip(ip)
```

Arguments

`ip`	(`pkg_installation_plan`) object to execute

Value

`pkg_installation_plan` object invisibly

Index