Package 'yamaa'

November 17, 2026

Title Clinical Data Standardization Using 'YAML' Specifications

Version 0.6.2

Description Builds standardized clinical trial datasets using 'YAML' specifications. Supports the Study Data Tabulation Model (SDTM) and Analysis Data Model (ADaM), with specification checks, data derivations, and structured diagnostics. Generates submission metadata in 'Define-XML' format.

License MIT + file LICENSE

URL https://elong0527.github.io/yamaa/, https://github.com/elong0527/yamaa

BugReports https://github.com/elong0527/yamaa/issues

Copyright The yamaa contributors; for the bundled and linked third-party code and data, the holders that inst/AUTHORS lists and that inst/LICENSE-THIRD-PARTY.txt, inst/LICENSE-READSTAT.txt, inst/LICENSE-UNICODE.txt and inst/LICENSE-YAML.txt name.

Encoding UTF-8

Depends R (>= 4.2.0)

Imports utils

Suggests arrow (>= 23.0.1), jsonlite (>= 2.0.0), knitr (>= 1.42), rmarkdown (>= 2.20)

VignetteBuilder knitr

SystemRequirements Cargo (Rust's package manager) (>= 1.82.0), rustc (>= 1.82.0)

NeedsCompilation yes

Author Yilong Zhang [aut, cre, cph], Ming-Chun Chen [ctb], Peng Zhang [ctb], Bingjun Wang [ctb], Jingwei Xiong [ctb], The authors of the bundled Rust crates [ctb, cph] (listed in

inst/AUTHORS),
Evan Miller and the ReadStat authors [ctb, cph] (bundled ReadStat C sources, inst/LICENSE-READSTAT.txt)

Maintainer Yilong Zhang <elong0527@gmail.com>

Repository CRAN

Date/Publication 2026-11-17 12:00:02 UTC

Contents

yamaa_conditions 2
yamaa_info 3
yamaa_lint 4
yamaa_project 5
yamaa_read_odm 8
yamaa_schema 9
yamaa_selftest 10
yamaa_spec 11

Index **14**

`yamaa_conditions` *The conditions of this release and their history*

Description

Returns the condition catalogue: every condition the engine may report in an issue, and every condition a release retired, with what a program needs to match it. The catalogue is part of the installed engine, so nothing is read from disk, and the same table comes from every host. The file `yaml/conditions.yaml` of the source repository at a release's tag holds the same catalogue.

Usage

```
yamaa_conditions()
```

Value

A data frame with one row per condition, in name order, and these columns:

`condition` The condition, as an issue's `condition` column names it.

`severity` The severity of its issues: `"error"` for every condition.

`phases` A list column: the phases that may report the condition, as a character vector; empty for a retired condition.

`rules` A list column: the rule cards that its issues cite, such as `"VALUE-3"`, in card order; empty for a condition whose issues cite no rule and for a retired condition.

`message` The template its issues' messages are written from; `NA` for a retired condition. Messages may change in any release, so match on `condition` and `rule`, never on the message.

`renamed_from` The name the condition had before a release renamed it, or `NA`.

`retired_in` The release that retired the condition, such as `"1.0.0"`, or `NA` for a condition the engine may report.

Examples

```
catalogue <- yamaa_conditions()
nrow(catalogue)
catalogue[!is.na(catalogue$renamed_from), c("condition", "renamed_from")]
catalogue$condition[!is.na(catalogue$retired_in)]
```

---

yamaa_info *What the engine is and where it runs*

Description

Returns the facts that identify the installed engine: its version, the language versions it reads, the features it was built with, the host and the platform. A selftest's `info` holds the same record, and the command line prints it for `yamaa --version`. It reads and writes no file.

Usage

```
yamaa_info()
```

Value

A named list, in this order:

`version` The engine version, the same in every host; a release candidate reads like `"1.0.0-rc.1"`.

`language_versions` Every `yamaa:` language version the engine reads, oldest first.

`features` The optional features the engine was built with, sorted, among `"odm"`, `"parquet"`, `"readstat"` and `"yaml"`; every released package has all four.

`host` `"r"`.

`host_version` The version of R, as `getRversion()` gives it.

`platform` The operating system and architecture of the engine, such as `"linux-x86_64"`.

`packageVersion("yamaa")` stays the version of the package, which keeps `1.0.0` while the release candidates carry their own engine version.

Examples

```
info <- yamaa_info()
info$version
info$platform
```

yamaa_lint *Lint yamaa files against the style guide*

Description

Reports how each yamaa file departs from the written form the yamaa style guide fixes: field order, blank lines, line width, the canonical spellings of a 1.0 file and the authoring findings. A finding describes the text, never its meaning, and a file runs as its canonical form does.

Usage

```
yamaa_lint(paths, fix = FALSE)
```

Arguments

paths	Paths of yamaa files. A directory names the yamaa files of every file type beneath it, outside hidden directories.
fix	Whether to apply the proved fixes first. A file is replaced only when the fixed text reads as the same values with the same comments, or as those values with exactly the canonical rewrite a finding names.

Details

A comment # yamaa-lint: allow <finding> -- <reason> directly above a field or an entry suppresses the named findings within it.

Value

A data frame with one row per finding that remains and the columns file (the absolute path, with forward slashes), line, column, finding, spec_path and message. A file that cannot be read or parsed, or a directory that cannot be listed, raises a condition of class yamaa_error, and nothing is written.

Examples

```
study <- tempfile("yamaa-")
dir.create(study)
spec <- file.path(study, "dm.yaml")
writeLines('yamaa: "1.0"
type: spec
domain: DM
input: {RAW: {path: input.csv, types: {USUBJID: str, AGE: int}}}
keys: [USUBJID]
columns:
  - {name: USUBJID, type: str, derivation: RAW.USUBJID}
  - {name: AGE, type: int, derivation: RAW.AGE}
output: {path: dm.csv, columns: [USUBJID, AGE]}
', spec)
```

```
findings <- yamaa_lint(study)
findings[, c("line", "finding", "message")]
remaining <- yamaa_lint(spec, fix = TRUE)
remaining[, c("line", "finding", "message")]

unlink(study, recursive = TRUE)
```

`yamaa_project` *A project file, read only when a method runs*

Description

Holds the path of a `project.yaml`: the study's specifications, codelists, submission metadata, function files and per-language locks. Nothing is read until a method runs.

Usage

```
yamaa_project(path)

## S3 method for class 'yamaa_project_run'
x[[i, ...]]

## S3 method for class 'yamaa_project'
print(x, ...)

## S3 method for class 'yamaa_project'
format(x, ...)

## S3 method for class 'yamaa_project_run'
print(x, ...)

## S3 method for class 'yamaa_project_run'
format(x, ...)

## S3 method for class 'yamaa_project_run'
names(x)

## S3 method for class 'yamaa_project_run'
length(x)

## S3 method for class 'yamaa_member_run'
print(x, ...)

## S3 method for class 'yamaa_member_run'
format(x, ...)
```

```
## S3 method for class 'yamaa_member_run'
names(x)

## S3 method for class 'yamaa_member_run'
length(x)

## S3 method for class 'yamaa_submission'
print(x, ...)

## S3 method for class 'yamaa_submission'
format(x, ...)

## S3 method for class 'yamaa_submission'
names(x)

## S3 method for class 'yamaa_submission'
length(x)
```

Arguments

`path`	Path of a `project.yaml`.
`x`	A project, a project run, one specification's run of a project run, or a submission.
`i`	The `domain` name of a specification the run built.
`...`	Unused.

Value

An object with a read-only field and four methods:

`$names` The `domain` names of the listed specifications, in the order that a run of every section builds them, each after the ones it reads from, read without study data. A project file that cannot be read, or that lists a name twice, raises.

`$spec(name)` The listed specification whose `domain` is `name`, with the project's codelists and functions.

`$check()` Static findings for every listed specification, with `$status` and `$issues`.

`$run(section = NULL)` Builds every specification of one section (`"sdtm"`, `"adam"` or `"send"`), or of all sections. The project run has `$status`, `"succeeded"` when every specification succeeded, `$names`, `$issues`, `$report` and `$save()`, which writes each declared report file and then each output dataset in run order, only the reports when a specification failed; `runs[["ADSL"]]` is one specification's run, with `$status`, `$output`, `$issues` and `$report`. Its status is `"skipped"` when a specification that it reads from did not complete.

`$submission(section = NULL, created = NULL)` define.xml, and a Dataset-JSON or SAS Transport version 5 file per dataset where a section sets `datasets: json` or `datasets: xpt`, with `$status`, `$created`, the ISO 8601 creation time the documents carry (the `created` given, or else the time read once for every section), `$issues` and `$save()`, which writes each section's transport files and then its define.xml.

`$check()`, `$run()` and `$submission()` also take `progress`, a function called with one named list per event, as `yamaa_spec` describes: each specification of a check or run is named by its `domain`, and each section of a submission is one `"composition"` phase. A `progress` function that signals an error is called no more: the call checks, builds or composes nothing further, and its result fails with one `engine_rejected` issue whose code is `host_progress_failure` and refuses its save. A check, a project run, one specification's run and a submission hold `$timings`, a data frame of each phase the call ran; no saved file holds them.

Each `$save()` returns the absolute paths it wrote invisibly, in publication order. A failure raises a condition of class `yamaa_error` whose `issues` data frame says why; for a refused `$save()`, its `paths` lists the files written before the refusal.

`format()` returns the lines that the engine renders for an object, the same words that Python and the command line print, and `print()` writes them and returns the object invisibly. A project prints `yamaa project <path>`, since it has read nothing. A project run prints its status, its number of specifications and its issue count, then one line per specification with its status, size and issue count, then its first issue; one specification's run prints as a run does. A submission prints its status, its define.xml and transport files and its issue count.

`names()` of a project run gives the names of its specifications, as `$names` does, and `length()` counts them. `names()` of one specification's run or of a submission lists its fields and methods in the order above, starting with `status`, and `length()` counts them; `ls()` lists the same names, sorted.

Examples

```
study <- tempfile("yamaa-")
dir.create(study)
writeLines(c("USUBJID,AGE", "01,35", "02,42"), file.path(study, "input.csv"))
writeLines('yamaa: "1.0"
type: spec
model: SDTM
domain: DM
keys: [USUBJID]
input: {RAW: {path: input.csv, types: {USUBJID: str, AGE: int}}}
output: {path: dm.csv, columns: [USUBJID, AGE]}

columns:
  - {name: USUBJID, type: str, derivation: RAW.USUBJID}
  - {name: AGE, type: int, derivation: RAW.AGE}
', file.path(study, "dm.yaml"))
writeLines('yamaa: "1.0"
type: project
sdtm: {specs: [dm.yaml]}
', file.path(study, "project.yaml"))

project <- yamaa_project(file.path(study, "project.yaml"))
project$names
project$check()
runs <- project$run("sdtm")
runs
runs[["DM"]]$output
basename(runs$save())
```

```
unlink(study, recursive = TRUE)
```

`yamaa_read_odm` *Read the clinical items of an ODM XML file*

Description

Reads an ODM 1.3 or 2.0 XML file, or one member of a TAR archive, into one row per clinical item.

Usage

```
yamaa_read_odm(source, archive_member = NULL, max_archive_members = 10000,
               max_expanded_bytes = 268435456, max_xml_depth = 64)
```

Arguments

`source`	Path of an ODM XML file or TAR archive.
`archive_member`	Name of the XML member to read from a TAR, or `NULL` to select the only XML member.
`max_archive_members`	Most members a TAR may hold.
`max_expanded_bytes`	Most bytes the input may expand to.
`max_xml_depth`	Deepest element nesting accepted.

Value

A data frame with one row per clinical item. A refusal raises a condition of class `yamaa_error`.

Examples

```
odm <- tempfile(fileext = ".xml")
writeLines('<?xml version="1.0" encoding="UTF-8"?>
<ODM xmlns="http://www.cdisc.org/ns/odm/v1.3" ODMVersion="1.3" FileOID="F.1">
  <ClinicalData StudyOID="S.1" MetaDataVersionOID="M.1">
    <SubjectData SubjectKey="01">
      <StudyEventData StudyEventOID="SE.SCREEN">
        <FormData FormOID="F.DM">
          <ItemGroupData ItemGroupOID="IG.DM">
            <ItemData ItemOID="IT.DM.AGE" Value="35"/>
            <ItemData ItemOID="IT.DM.SEX" Value="F"/>
          </ItemGroupData>
        </FormData>
      </StudyEventData>
    </SubjectData>
  </ClinicalData>
</ODM>
```

```
', odm)

items <- yamaa_read_odm(odm)
items[, c("SubjectKey", "ItemOID", "Value")]

unlink(odm)
```

---

yamaa_schema *The JSON Schema of a yamaa file type*

Description

Returns the JSON Schema that a file of one type is checked against, so that an editor can validate and complete the file as it is typed. The schema is part of the installed engine, so nothing is read from disk, and the same text comes from every host.

Usage

```
yamaa_schema(file_type)
```

Arguments

file_type	The file type, as a file writes it in its type header: "spec", "project", "codelists" or "function".

Details

A specification file may be one layer of an inheritance chain, so the "spec" schema checks what one layer may write, and its $defs/root_class checks a complete specification. Every field carries its kind as x-yamaa-kind and its schema-notation type as x-yamaa-type.

Value

One character string: the JSON Schema as canonical JSON text, on one line with its members in key order. Any other file type raises a condition of class yamaa_error whose issue names the condition unexpected_file_type.

Examples

```
schema <- yamaa_schema("spec")
nchar(schema)
substr(schema, 1, 40)
```

`yamaa_selftest` *Run the bundled selftest cases through the installed engine*

Description

Runs the selftest cases that the package carries, benchmarks with their independently authored expected files, through the installed engine and compares each result with its expected files. It reads and writes no file and uses no network. Its report is the installation and operational qualification evidence of an installation.

Usage

```
yamaa_selftest()

## S3 method for class 'yamaa_selftest'
print(x, ...)

## S3 method for class 'yamaa_selftest'
format(x, ...)

## S3 method for class 'yamaa_selftest'
names(x)

## S3 method for class 'yamaa_selftest'
length(x)
```

Arguments

`x`	A selftest result.
`...`	Unused.

Value

An object with three read-only fields:

`$status` `"failed"` when a case failed, and `"succeeded"` otherwise.

`$cases` A data frame with one row per case and the columns `case`, `status` and `message`. A case's status is `"succeeded"`, `"failed"`, or `"skipped"` when this build does not compile a feature the case needs; its message says why it failed or was skipped, and is `NA` when it passed.

`$info` A named list: the engine `version`, the `language_versions` it reads, its compiled `features`, the `host` (`"r"`), the `host_version` and the `platform`.

`format()` returns the lines of the report, and `print()` writes them and returns the result invisibly. The first line reads `yamaa selftest: <status>, <p> of <c> cases passed`. The first failed case and the adapters that no case covered follow, then the engine version, the language versions, the host and the platform. `names()` lists the three fields in that order, and `length()` counts them.

Examples

```
result <- yamaa_selftest()
result
result$cases
```

---

yamaa_spec *One specification, read only when a method runs*

Description

Holds the path of one specification. Nothing is read until a method runs, and each method rereads the files.

Usage

```
yamaa_spec(path)

## S3 method for class 'yamaa_spec'
print(x, ...)

## S3 method for class 'yamaa_spec'
format(x, ...)

## S3 method for class 'yamaa_check'
print(x, ...)

## S3 method for class 'yamaa_check'
format(x, ...)

## S3 method for class 'yamaa_check'
names(x)

## S3 method for class 'yamaa_check'
length(x)

## S3 method for class 'yamaa_run'
print(x, ...)

## S3 method for class 'yamaa_run'
format(x, ...)

## S3 method for class 'yamaa_run'
names(x)

## S3 method for class 'yamaa_run'
length(x)
```

Arguments

path	Path of a specification file.
x	A specification, a check or a run.
...	Unused.

Value

An object with three methods:

`$check(parent = NULL)` A check: `$status` is `"failed"` when the check found a problem, and `"succeeded"` otherwise, and the `$issues` data frame lists the problems found without reading study data or running project code. With `parent`, the path of another specification, the specification is checked as a fragment whose parent is `parent`, as it would compose in a file that names `parent` as its `parent`, includes the fragment and writes nothing else; a specification that declares `parent` or `include` is no fragment, and the call fails with a `yamaa_error` condition.

`$resolve()` The specification after its layers are composed, as a named list.

`$run()` A run: `$status` is `"succeeded"` when the run built its dataset with no issue, and `"failed"` otherwise, `$output` is the dataset (`NULL` when the run failed), `$issues` what is wrong, `$report` what the run read, counted and checked, and `$save()` writes the declared report file, then the output dataset, and returns their absolute paths invisibly, in that order. A failed run writes only its report file.

`$run(stream = TRUE)` builds the dataset as its output is read, and reads nothing until then: `$output` is an Arrow C stream, an external pointer of class `nanoarrow_array_stream`, which nanoarrow and arrow read, one batch per pull; the first pull builds the run. The other fields give the run once its stream has ended, and run it whole when nothing has read the output, whose stream then reads the dataset it keeps. Releasing the stream before its last batch cancels the run, which then saves nothing.

`$check()` and `$run()` also take `progress`, a function called on the calling thread with one named list per event as each phase starts and ends: `name`, `phase`, `kind` (`"start"`, `"end"`, `"partition"` or `"batch"`), `outcome`, `number` and `rows`, the last three `NULL` where the event has none. A `progress` function that signals an error is called no more, and the check or run then fails with one `engine_rejected` issue whose code is `host_progress_failure`; such a run saves nothing. A check and a run hold `$timings`, a data frame of each phase the call ran, with the columns `name`, `phase`, `started_ms`, `elapsed_ms` and `outcome`; no saved file holds them.

A failure raises a condition of class `yamaa_error` whose `issues` data frame says why; for a refused `$save()`, its `paths` lists the files written before the refusal.

`format()` returns the lines that the engine renders for an object, the same words that Python and the command line print, and `print()` writes them and returns the object invisibly. A specification prints `yamaa spec <path>`, with the path as given, since it has read nothing. A check prints `yamaa check of <path>: <status>, <n> issues`, and a run `yamaa run of <dataset>: <status>, <size>, <n> issues`, where the size is `<r> rows x <c> columns` or `no output`; either then prints its first issue on a line of its own.

`names()` of a check or a run lists its fields and methods in the order above, starting with `status`, and `length()` counts them; `ls()` lists the same names, sorted.

Examples

```
study <- tempfile("yamaa-")
dir.create(study)
writeLines(c("USUBJID,AGE", "01,35", "02,42"), file.path(study, "input.csv"))
writeLines('yamaa: "1.0"
type: spec
domain: DM
keys: [USUBJID]
input: {RAW: {path: input.csv, types: {USUBJID: str, AGE: int}}}
output: {path: dm.csv, columns: [USUBJID, AGE]}

columns:
  - {name: USUBJID, type: str, derivation: RAW.USUBJID}
  - {name: AGE, type: int, derivation: RAW.AGE}
', file.path(study, "dm.yaml"))

spec <- yamaa_spec(file.path(study, "dm.yaml"))
spec$check()
run <- spec$run()
run
run$output
basename(run$save())

unlink(study, recursive = TRUE)
```

Index

[[.yamaa_project_run *(yamaa_project)*, 5

format.yamaa_check *(yamaa_spec)*, 11
format.yamaa_member_run *(yamaa_project)*, 5
format.yamaa_project *(yamaa_project)*, 5
format.yamaa_project_run *(yamaa_project)*, 5
format.yamaa_run *(yamaa_spec)*, 11
format.yamaa_selftest *(yamaa_selftest)*, 10
format.yamaa_spec *(yamaa_spec)*, 11
format.yamaa_submission *(yamaa_project)*, 5

length.yamaa_check *(yamaa_spec)*, 11
length.yamaa_member_run *(yamaa_project)*, 5
length.yamaa_project_run *(yamaa_project)*, 5
length.yamaa_run *(yamaa_spec)*, 11
length.yamaa_selftest *(yamaa_selftest)*, 10
length.yamaa_submission *(yamaa_project)*, 5

names.yamaa_check *(yamaa_spec)*, 11
names.yamaa_member_run *(yamaa_project)*, 5
names.yamaa_project_run *(yamaa_project)*, 5
names.yamaa_run *(yamaa_spec)*, 11
names.yamaa_selftest *(yamaa_selftest)*, 10
names.yamaa_submission *(yamaa_project)*, 5

print.yamaa_check *(yamaa_spec)*, 11
print.yamaa_member_run *(yamaa_project)*, 5
print.yamaa_project *(yamaa_project)*, 5
print.yamaa_project_run *(yamaa_project)*, 5
print.yamaa_run *(yamaa_spec)*, 11
print.yamaa_selftest *(yamaa_selftest)*, 10
print.yamaa_spec *(yamaa_spec)*, 11
print.yamaa_submission *(yamaa_project)*, 5

yamaa_conditions, 2
yamaa_info, 3
yamaa_lint, 4
yamaa_project, 5
yamaa_read_odm, 8
yamaa_schema, 9
yamaa_selftest, 10
yamaa_spec, *7*, 11