Package ‘ruminate’

January 10, 2025

Title A Pharmacometrics Data Transformation and Analysis Tool

Version 0.3.1

Description Exploration of pharmacometrics data involves both general tools (transformation and plotting) and specific techniques (non-compartmental analysis). This kind of exploration is generally accomplished by utilizing different packages. The purpose of 'ruminate' is to create a 'shiny' interface to make these tools more broadly available while creating reproducible results.

License BSD_2_clause + file LICENSE

BugReports `https://github.com/john-harrold/ruminate/issues`

URL `https://ruminate.ubiquity.tools/`

Encoding UTF-8

Depends R (>= 4.2.0)

Imports digest, dplyr, DT, flextable, formods (>= 0.2.0), ggplot2, onbrand (>= 1.0.3), PKNCA (>= 0.10.2), plotly, rhandsontable, rlang, shiny, shinyAce, shinyWidgets, stats, stringr, tidyr, utils, yaml, zip

Suggests babelmixr2, cli, clipr, gridExtra, knitr, nlmixr2lib(>= 0.3.0), nonmem2rx, nlmixr2(>= 3.0.0), prompter, rmarkdown, readxl, rxode2 (>= 3.0.0), shinybusy, shinydashboard, testthat (>= 3.0.0), ubiquity

RoxygenNote 7.3.2

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

Author John Harrold [aut, cre] (<`https://orcid.org/0000-0003-2052-4373`>)

Maintainer John Harrold <`john.m.harrold@gmail.com`>

Repository CRAN

Date/Publication 2025-01-10 15:50:06 UTC

Contents

`apply_route_map` *Applies Route Mapping to Dataset*

Description

Used to convert nonstandard dose route values (i.e. "IV") to standard values ("intravascular").

Usage

```
apply_route_map(route_map = list(), route_col = NULL, DS = NULL)
```

Arguments

`route_map`	List with names corresponding to the route replacement and a vector of regular expressions to match.
`route_col`	Column name with the route data.
`DS`	Dataframe containing the dataset.

Value

Dataset with the route mapping applied.

Examples

```
if(system.file(package="readxl") !=""){
library(readxl)
#loading a dataset
data_file =  system.file(package="formods","test_data","TEST_DATA.xlsx")
myDS = readxl::read_excel(path=data_file, sheet="DATA")

 route_map = list(
   intravascular = c("^(?i)iv$"),
   extravascular = c("^(?i)sc$", "^(?i)oral")
 )

utils::head(myDS[["ROUTE"]])

myDS = apply_route_map(route_map = route_map,
                       route_col = "ROUTE",
                       DS        = myDS)

utils::head(myDS[["ROUTE"]])
}
```

`CTS_add_covariate` *Add Covariate to Elemetnt*

Description

Takes the ui elements in the module state and processes the covariate elements for addition.

Usage

```
CTS_add_covariate(state, element)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`
`element`	Element list from `CTS_fetch_current_element()`

Details

This depends on the following UI values in the state

- `state[["CTS"]][["ui"]][["covariate_value"]]`
- `state[["CTS"]][["ui"]][["covariate_type_selected"]]`
- `state[["CTS"]][["ui"]][["selected_covariate"]]`

Value

Element with the results of adding the covariate. The `cares` list element can be used to determine the exit status of the function.

- COV_IS_GOOD If TRUE if the covariate was good and added, and FALSE if there were any issues.
- msgs Vector of messages.

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
                        id_ASM          = id_ASM,
                        id_MB           = id_MB,
                        input           = input,
                        session         = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
```

```
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)
```

```
# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_add_rule` *Add Rule to Element*

Description

Takes the ui elements in the state and element and attempts to add a rule.

Usage

```
CTS_add_rule(state, element)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`
`element`	Element list from `CTS_fetch_current_element()`

Details

This depends on the following UI values in the state and element

- `state[["CTS"]][["ui"]][["rule_name"]]`
- `state[["CTS"]][["ui"]][["rule_condition"]]`
- `state[["CTS"]][["ui"]][["rule_type"]]`

  - For rule type "dose"
    - `state[["CTS"]][["ui"]][["action_dosing_state"]]`
    - `state[["CTS"]][["ui"]][["action_dosing_values"]]`
    - `state[["CTS"]][["ui"]][["action_dosing_times"]]`
    - `state[["CTS"]][["ui"]][["action_dosing_durations"]]`
  - For rule type "set state"
    - `state[["CTS"]][["ui"]][["action_set_state_state"]]`
    - `state[["CTS"]][["ui"]][["action_set_state_values"]]`
  - For rule type "manual code"
    - `state[["CTS"]][["ui"]][["action_manual_code"]]`

Value

Element with the results of adding the rule. The `rares` list element can be used to determine the exit status of the function.

- RULE_IS_GOOD If true it indicates that the pieces of the rule from the UI check out.
- RULE_UPDATED If RULE_IS_GOOD and RULE_UPDATED is true then a previous rule definition was overwritten. If RULE_IS_GOOD is TRUE and RULE_UPDATED is FALSE then a new rule was added.
- notify_text Text for notify message
- notify_id Notification ID
- notify_type Notification type
- msgs Vector of messages.

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
```

```
                            id_ASM          = id_ASM,
                            id_MB           = id_MB,
                            input           = input,
                            session         = session,
                            FM_yaml_file    = FM_yaml_file,
                            MOD_yaml_file   = MOD_yaml_file,
                            react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
```

```
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[[”CTS”]][[”ui”]][[”covariate_value”]]            = ”70, .1”
state[[”CTS”]][[”ui”]][[”covariate_type_selected”]]    = ”cont_lognormal”
state[[”CTS”]][[”ui”]][[”selected_covariate”]]         = ”WT”
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_append_report` *Append Report Elements*

Description

Appends report elements to a formods report.

Usage

```
CTS_append_report(state, rpt, rpttype, gen_code_only = FALSE)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`
`rpt`	Report with the current content of the report which will be appended to in this function. For details on the structure see the documentation for `onbrand::template_details()`
`rpttype`	Type of report to generate (supported "xlsx", "pptx", "docx").
`gen_code_only`	Boolean value indicating that only code should be generated (`FALSE`).

Value

list containing the following elements

- isgood: Return status of the function.
- hasrptele: Boolean indicator if the module has any reportable elements.
- code: Code to generate reporting elements.
- msgs: Messages to be passed back to the user.
- rpt: Report with any additions passed back to the user.

See Also

`formods::FM_generate_report()` and `onbrand::template_details()`

`CTS_change_source_model` *Change the Source Model*

Description

Takes the ui elements in the state and element and processes any changes to the source model and updates the element accordingly.

Usage

```
CTS_change_source_model(state, element)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`
`element`	Element list from `CTS_fetch_current_element()`

Details

This depends on the following UI values in the state.

- `state[["CTS"]][["ui"]][["source_model"]]`

Value

Element with the necessary changes to the source model.

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
                        id_ASM          = id_ASM,
                        id_MB           = id_MB,
                        input           = input,
                        session         = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
```

```
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_del_current_element`

Deletes Current cohort

Description

Takes a CTS state and deletes the current cohort. If that is the last element, then a new default will be added.

Usage

```
CTS_del_current_element(state)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`

Value

CTS state object with the current cohort deleted.

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id             = id,
                        id_ASM         = id_ASM,
                        id_MB          = id_MB,
                        input          = input,
                        session        = session,
```

```
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
```

```
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_fetch_code` *Fetch Module Code*

Description

Fetches the code to generate results seen in the app

Usage

```
CTS_fetch_code(state)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`

Value

Character object vector with the lines of code

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
                        id_ASM          = id_ASM,
                        id_MB           = id_MB,
                        input           = input,
                        session         = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
```

```
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_fetch_current_element` *Fetches Current cohort*

Description

Takes a CTS state and returns the current active cohort

Usage

```
CTS_fetch_current_element(state)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`

Value

List containing the details of the active data view. The structure of this list is the same as the structure of `state$CTS$elements` in the output of `CTS_fetch_state()`.

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id             = id,
                        id_ASM         = id_ASM,
                        id_MB          = id_MB,
                        input          = input,
                        session        = session,
```

```
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
```

```
    state   = state,
    element = current_ele)

  # This will extract the code for the current module
  code = CTS_fetch_code(state)
  code

  # This will update the checksum of the module state
  state = CTS_update_checksum(state)


  # Access the datasets generated from simulations
  ds = CTS_fetch_ds(state)

  # CTS_add_covariate
  state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
  state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
  state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
  current_ele = CTS_add_covariate(state, current_ele)

  # Creates a new empty element
  state = CTS_new_element(state)

  # Delete the current element
  state = CTS_del_current_element(state)
  }
```

`CTS_fetch_ds` *Fetch Clinical Trial Simulator Module Datasets*

Description

Fetches the datasets produced by the module. For each cohort this will be the simulation timecourse and the event table

Usage

```
CTS_fetch_ds(state)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`

Value

Character object vector with the lines of code

list containing the following elements

- isgood: Return status of the function.

- hasds: Boolean indicator if the module has any datasets
- msgs: Messages to be passed back to the user.
- ds: List with datasets. Each list element has the name of the R-object for that dataset. Each element has the following structure:
  - label: Text label for the dataset
  - MOD_TYPE: Short name for the type of module.
  - id: module ID
  - DS: Dataframe containing the actual dataset.
  - DSMETA: Metadata describing DS
  - code: Complete code to build dataset.
  - checksum: Module checksum.
  - DSchecksum: Dataset checksum.

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
                        id_ASM          = id_ASM,
                        id_MB           = id_MB,
                        input           = input,
                        session         = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"
```

```
# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                  = "0"
current_ele[["ui"]][["cts_config_nsteps"]]            = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]            = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                 = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]       = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]      = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]       = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]   = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                 = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]             = "1"
current_ele[["ui"]][["dvcols"]]            = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]              = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)
```

```
# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_fetch_sc_meta` *Fetches Simulation Parameter Meta Information*

Description

This provides meta information about simulatino options. This includes option names, text descriptions, ui_names used, etc.

Usage

```
CTS_fetch_sc_meta(
  MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")
)
```

Arguments

`MOD_yaml_file`	Module configuration file with MC as main section.

Value

List with the following elements:

- config List from the YAML->MC->sim_config.
- summary: Dataframe with elements of config in tabular format.
- ui_config Vector of all the ui_ids for configuration options.

Examples

```
CTS_fetch_sc_meta()
```

`CTS_fetch_state` *Fetch Clinical Trial Simulator State*

Description

Merges default app options with the changes made in the UI

Usage

```
CTS_fetch_state(
  id,
  id_ASM,
  id_MB,
  input,
  session,
  FM_yaml_file,
  MOD_yaml_file,
  react_state
)
```

Arguments

`id`	Shiny module ID
`id_ASM`	ID string for the app state management module used to save and load app states
`id_MB`	An ID string that corresponds with the ID used to call the MB modules
`input`	Shiny input variable
`session`	Shiny session variable
`FM_yaml_file`	App configuration file with FM as main section.
`MOD_yaml_file`	Module configuration file with MC as main section.
`react_state`	Variable passed to server to allow reaction outside of module (`NULL`)

Value

list containing the current state of the app including default values from the yaml file as well as any changes made by the user. The list has the following structure:

- yaml: Full contents of the supplied yaml file.
- MC: Module components of the yaml file.
- CTS:
  - isgood: Boolean object indicating if the file was successfully loaded.
  - checksum: This is an MD5 sum of the contents element and can be used to detect changes in the state.
- MOD_TYPE: Character data containing the type of module `"CTS"`
- id: Character data containing the module id module in the session variable.
- FM_yaml_file: App configuration file with FM as main section.
- MOD_yaml_file: Module configuration file with MC as main section.

Examples

```
# Within shiny both session and input variables will exist,
# this creates examples here for testing purposes:
sess_res = MB_test_mksession()
session = sess_res$session
input   = sess_res$input

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

# Creating an empty state object
state = CTS_fetch_state(id              = "CTS",
                       id_ASM          = "ASM",
                       id_MB           = "MB",
                       input           = input,
                       session         = session,
                       FM_yaml_file    = FM_yaml_file,
                       MOD_yaml_file   = MOD_yaml_file,
                       react_state     = NULL)
```

`CTS_init_element_model`

Initializes Cohort When Model Changes

Description

When a source model changes this will update information about that model like the default dvcols and selection information about the dvcols

Usage

```
CTS_init_element_model(state, element)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`
`element`	Element list from `CTS_fetch_current_element()`

Value

CTS state object with the current cohort ui elements initialized based on the current model selected

---

`CTS_init_state` *Initialize CTS Module State*

Description

Creates a list of the initialized module state

Usage

```
CTS_init_state(FM_yaml_file, MOD_yaml_file, id, id_MB, session)
```

Arguments

`FM_yaml_file`	App configuration file with FM as main section
`MOD_yaml_file`	Module configuration file with MC as main section
`id`	ID string for the module
`id_MB`	An ID string that corresponds with the ID used to call the MB modules
`session`	Shiny session variable

Value

list containing an empty CTS state

Examples

```
# Within shiny both session and input variables will exist,
# this creates examples here for testing purposes:
sess_res = MB_test_mksession()
session = sess_res$session
input   = sess_res$input

state = CTS_init_state(
   FM_yaml_file  = system.file(package = "formods",
                               "templates",
                               "formods.yaml"),
   MOD_yaml_file = system.file(package = "ruminate",
                               "templates",
                               "CTS.yaml"),
   id             = "CTS",
   id_MB          = "MB",
   session        = session)

state
```

`CTS_mk_preload` *Make List of Current CTS State*

Description

Reads in the app state from yaml files.

Usage

```
CTS_mk_preload(state)
```

Arguments

`state`	CTS state object

Value

list with the following elements

- isgood: Boolean indicating the exit status of the function.
- msgs: Messages to be passed back to the user.
- yaml_list: Lists with preload components.

Examples

```
sess_res = CTS_test_mksession()
state = sess_res$state
res = CTS_mk_preload(state)
```

`CTS_new_element` *New Clinical Trial Simulation Cohort*

Description

Appends a new empty cohort to the CTS state object and makes this new cohort the active cohort.

Usage

```
CTS_new_element(state)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`

Value

CTS state object containing a new cohort and that cohort is set as the current active cohort. See the help for `CTS_fetch_state()` for ===ELEMENT== format.

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
                        id_ASM          = id_ASM,
                        id_MB           = id_MB,
                        input           = input,
                        session         = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
```

```
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
```

```
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_plot_element` *Plots the Specified Element*

Description

Takes a CTS state and element and simulates the current set of rules.

Usage

```
CTS_plot_element(state, element)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`
`element`	Element list from `CTS_fetch_current_element()`

Value

Simulation element with plot results stored in the '`"plotres"` element.

- isgood Boolean value indicating the state of the figure generation code.
- msgs Any messages to be passed to the user.
- capture Captured figure generation output from `plot_sr_tc()`

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()
```

```
# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
                        id_ASM          = id_ASM,
                        id_MB           = id_MB,
                        input           = input,
                        session         = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)
```

```
# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_preload` *Preload Data for CTS Module*

Description

Populates the supplied session variable with information from list of sources.

Usage

```
CTS_preload(
  session,
  src_list,
  yaml_res,
```

```
  mod_ID = NULL,
  react_state = list(),
  quickload = FALSE
)
```

Arguments

`session`	Shiny session variable (in app) or a list (outside of app)
`src_list`	List of preload data (all read together with module IDs at the top level)
`yaml_res`	List data from module yaml config
`mod_ID`	Module ID of the module being loaded.
`react_state`	Reactive shiny object (in app) or a list (outside of app) used to trigger reactions.
`quickload`	Logical `TRUE` to load reduced analysis `FALSE` to load the full analysis

Value

list with the following elements

- isgood: Boolean indicating the exit status of the function.
- msgs: Messages to be passed back to the user.
- session: Session object
- input: The value of the shiny input at the end of the session initialization.
- state: App state.
- react_state: The `react_state` components.

`CTS_Server` *Clinical Trial Simulator State Server*

Description

Server function for the Clinical Trial Simulator Shiny Module

Usage

```
CTS_Server(
  id,
  id_ASM = "ASM",
  id_MB = "MB",
  FM_yaml_file = system.file(package = "formods", "templates", "formods.yaml"),
  MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml"),
  deployed = FALSE,
  react_state = NULL
)
```

Arguments

`id`	An ID string that corresponds with the ID used to call the modules UI elements
`id_ASM`	ID string for the app state managment module used to save and load app states
`id_MB`	An ID string that corresponds with the ID used to call the MB modules
`FM_yaml_file`	App configuration file with FM as main section.
`MOD_yaml_file`	Module configuration file with MC as main section.
`deployed`	Boolean variable indicating whether the app is deployed or not.
`react_state`	Variable passed to server to allow reaction outside of module (`NULL`)

Value

UD Server object

`CTS_set_current_element` *Sets the Value for the Current cohort*

Description

Takes a CTS state and returns the current active cohort

Usage

```
CTS_set_current_element(state, element)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`
`element`	Element list from `CTS_fetch_current_element()`

Value

CTS state object with the current cohort set using the supplied value.

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]
```

```
id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
                        id_ASM          = id_ASM,
                        id_MB           = id_MB,
                        input           = input,
                        session         = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]              = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"
```

```
# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_simulate_element` *Simulates the Specified Element*

Description

Takes a CTS state and element and simulates the current set of rules.

Usage

```
CTS_simulate_element(state, element)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`
`element`	Element list from `CTS_fetch_current_element()`

Value

Simulation element with simulation results stored in the `"simres"` element.

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
                        id_ASM          = id_ASM,
                        id_MB           = id_MB,
                        input           = input,
                        session         = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
```

```
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
```

```
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)

# Delete the current element
state = CTS_del_current_element(state)
}
```

`CTS_sim_isgood` *Checks Simulation in Element for Goodness*

Description

Takes the supplied element and determines if the underlying simulation is in a good state or not.

Usage

```
CTS_sim_isgood(state, element)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`
`element`	Element list from `CTS_fetch_current_element()`

Value

List with the following elements:

- isgood: Boolean object indicating if the file was successfully loaded.
- msgs: Text description of failure.

`CTS_test_mksession` *Populate Session Data for Module Testing*

Description

Populates the supplied session variable for testing.

Usage

```
CTS_test_mksession(session = list(), full = FALSE)
```

Arguments

`session`	Shiny session variable (in app) or a list (outside of app)
`full`	Boolean indicating if the full test session should be created (`TRUE`) or a minimal test session should be created (`FALSE`, default)

Value

The CTS portion of the `all_sess_res` returned from `FM_app_preload`

See Also

`FM_app_preload`

Examples

```
session = shiny::MockShinySession$new()
sess_res = CTS_test_mksession(session=session)
```

`CTS_update_checksum` *Updates CTS Module Checksum*

Description

Takes a CTS state and updates the checksum used to trigger downstream updates

Usage

```
CTS_update_checksum(state)
```

Arguments

`state`	CTS state from `CTS_fetch_state()`

Value

CTS state object with the checksum updated

Examples

```
# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html
# None of this will work if rxode2 isn't installed:

library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){

# This will populate the session variable with the model building (MB) module
sess_res = MB_test_mksession()
session = sess_res[["session"]]

id     = "CTS"
id_ASM = "ASM"
id_MB  = "MB"
input  = list()

# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "CTS.yaml")

state = CTS_fetch_state(id              = id,
                        id_ASM          = id_ASM,
                        id_MB           = id_MB,
                        input           = input,
                        session         = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        react_state     = NULL)


# Fetch a list of the current element
current_ele = CTS_fetch_current_element(state)

# You can modify the element
current_ele[["element_name"]] = "A more descriptive name"

# Defining the source model
state[["CTS"]][["ui"]][["source_model"]] = "MB_obj_1_rx"
current_ele = CTS_change_source_model(state, current_ele)

# Single visit
current_ele[["ui"]][["visit_times"]]                 = "0"
current_ele[["ui"]][["cts_config_nsteps"]]           = "5"

# Creating a dosing rule
state[["CTS"]][["ui"]][["rule_condition"]]           = "time == 0"
state[["CTS"]][["ui"]][["rule_type"]]                = "dose"
```

```
state[["CTS"]][["ui"]][["action_dosing_state"]]      = "central"
state[["CTS"]][["ui"]][["action_dosing_values"]]     = "c(1)"
state[["CTS"]][["ui"]][["action_dosing_times"]]      = "c(0)"
state[["CTS"]][["ui"]][["action_dosing_durations"]]  = "c(0)"
state[["CTS"]][["ui"]][["rule_name"]]                = "Single_Dose"

# Adding the rule:
current_ele = CTS_add_rule(state, current_ele)

# Appending the plotting details as well
current_ele[["ui"]][["fpage"]]            = "1"
current_ele[["ui"]][["dvcols"]]           = "Cc"

# Reducing the number of subjects and steps to speed things up on CRAN
current_ele[["ui"]][["nsub"]]             = "2"
current_ele[["ui"]][["cts_config_nsteps"]] = "5"

# Putting the element back in the state forcing code generation
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# Now we pull out the current element, and simulate it
current_ele = CTS_fetch_current_element(state)
#current_ele = CTS_simulate_element(state, current_ele)

# Next we plot the element
current_ele = CTS_plot_element(state, current_ele)

# Now we save those results back into the state:
state = CTS_set_current_element(
  state   = state,
  element = current_ele)

# This will extract the code for the current module
code = CTS_fetch_code(state)
code

# This will update the checksum of the module state
state = CTS_update_checksum(state)


# Access the datasets generated from simulations
ds = CTS_fetch_ds(state)

# CTS_add_covariate
state[["CTS"]][["ui"]][["covariate_value"]]            = "70, .1"
state[["CTS"]][["ui"]][["covariate_type_selected"]]    = "cont_lognormal"
state[["CTS"]][["ui"]][["selected_covariate"]]         = "WT"
current_ele = CTS_add_covariate(state, current_ele)

# Creates a new empty element
state = CTS_new_element(state)
```

```
# Delete the current element
state = CTS_del_current_element(state)
}
```

`dose_records_builder` *Builds Dose Records Dataframe*

Description

Takes information about columns in dataset and constructs the dosing records.

Usage

```
dose_records_builder(
  NCA_DS = NULL,
  dose_from = NULL,
  col_id = NULL,
  col_time = NULL,
  col_ntime = NULL,
  col_route = NULL,
  col_dose = NULL,
  col_cycle = NULL,
  col_dur = NULL,
  col_evid = NULL,
  col_analyte = NULL,
  col_group = NULL
)
```

Arguments

`NCA_DS`	Dataset containing dosing records.
`dose_from`	Method of dose extraction either "cols" or "rows".
`col_id`	Name of column with subject ID.
`col_time`	Name of column with time since first dose.
`col_ntime`	Name of column with time since the last dose (required with `dose_from="cols"`).
`col_route`	Name of column with route information.
`col_dose`	Name of column with last dose given.
`col_cycle`	Name of column with dose cycle (required with `dose_from="cols"`).
`col_dur`	Name of column with dose duration.
`col_evid`	Name of column with event ID (required with `dose_from="rows"`).
`col_analyte`	Name of column with analyte (optional).
`col_group`	Names of columns with grouping information (optionl).

Value

list containing the following elements

- isgood: Return status of the function.
- msgs: Messages to be passed back to the user.
- dose_rec:

Examples

```
if(system.file(package="readxl") !=""){
library(dplyr)
library(readxl)
library(stringr)

# Example data file:
data_file =  system.file(package="formods","test_data","TEST_DATA.xlsx")

# Dataset formatted to extract dosing from columns
DS_cols = readxl::read_excel(path=data_file, sheet="DATA")        |>
 dplyr::filter(EVID == 0)                                    |>
 dplyr::filter(DOSE %in% c(3))                               |>
 dplyr::filter(str_detect(string=Cohort, "^MD"))             |>
 dplyr::filter(CMT == "C_ng_ml")

drb_res = dose_records_builder(
 NCA_DS     = DS_cols,
 dose_from  = "cols",
 col_id     = "ID",
 col_time   = "TIME_DY",
 col_ntime  = "NTIME_DY",
 col_route  = "ROUTE",
 col_cycle  = "DOSE_NUM",
 col_dose   = "DOSE",
 col_group  = "Cohort")

utils::head(drb_res$dose_rec)

# Dataset formatted to extract dosing from rows (records)
DS_rows = readxl::read_excel(path=data_file, sheet="DATA")        |>
 dplyr::filter(DOSE %in% c(3))                                   |>
 dplyr::filter(str_detect(string=Cohort, "^MD"))                 |>
 dplyr::filter(CMT %in% c("Ac", "C_ng_ml"))

drb_res = dose_records_builder(
 NCA_DS     = DS_rows,
 dose_from  = "rows",
 col_id     = "ID",
 col_time   = "TIME_DY",
 col_ntime  = "NTIME_DY",
 col_route  = "ROUTE",
 col_dose   = "AMT",
 col_evid   = "EVID",
```

```
 col_group  = "Cohort")

utils::head(drb_res$dose_rec)
}
```

`fetch_rxinfo` *Fetches Information from an rxode2 Object*

Description

This will provide information like parameter names, covriates, etc from an rxode2 object.

Usage

```
fetch_rxinfo(object)
```

Arguments

`object`	rxode2 model object An ID string that corresponds with the ID used to call the modules UI elements

Value

List with the following elements.

- isgood: Boolean variable indicating if the model is good.
- msgs: Any messages from parsing the model.
- elements: List with names of simulation elements:
  - covariates: Names of the covariates in the system.
  - parameters: Names of the parameters (subject level) in the system.
  - iiv: Names of the iiv parameters in the system.
  - states: Names of the states/compartments in the system.
- txt_info: Summary information in text format.
- list_info: Summary information in list format used with onbrand reporting.
- ht_info: Summary information in HTML formot.

Examples

```
library(formods)
library(ggplot2)

# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html

# None of this will work if rxode2 isn't installed:
if(is_installed("rxode2")){
library(rxode2)
```

```
set.seed(8675309)
rxSetSeed(8675309)

my_model = function ()
{
  description <- "One compartment PK model with linear clearance using differential equations"
    ini({
        lka <- 0.45
        label("Absorption rate (Ka)")
        lcl <- 1
        label("Clearance (CL)")
        lvc <- 3.45
        label("Central volume of distribution (V)")
        propSd <- c(0, 0.5)
        label("Proportional residual error (fraction)")
        etalcl ~ 0.1
    })
    model({
        ka <- exp(lka)
        cl <- exp(lcl + etalcl)
        vc <- exp(lvc)
        kel <- cl/vc
        d/dt(depot) <- -ka * depot
        d/dt(central) <- ka * depot - kel * central
        Cc <- central/vc
        Cc ~ prop(propSd)
    })
}

# This creates an rxode2 object
object  = rxode(my_model)

# If you want details about the parameters, states, etc
# in the model you can use this:
rxdetails = fetch_rxinfo(object)

rxdetails$elements

# Next we will create subjects. To do that we need to
# specify information about covariates:
nsub = 2
covs = list(
  WT         = list(type     = "continuous",
                    sampling = "log-normal",
                    values   = c(70, .15))
)

subs = mk_subjects(object = object,
                   nsub   = nsub,
                   covs   = covs)

head(subs$subjects)
```

```
rules = list(
  dose = list(
    condition = "TRUE",
    action    = list(
      type  = "dose",
      state     = "central",
      values    = "c(1)",
      times     = "c(0)",
      durations = "c(0)")
    )
)

# We evaulate the rules for dosing at time 0
eval_times =  0

# Stop 2 months after the last dose
output_times = seq(0, 56, 1)

# This runs the rule-based simulations
simres =
  simulate_rules(
    object        = object,
    subjects      = subs[["subjects"]],
    eval_times    = eval_times,
    output_times  = output_times,
    rules         = rules)

# First subject data:
sub_1 = simres$simall[simres$simall$id == 1, ]

# First subjects events
evall = as.data.frame(simres$evall)
ev_sub_1 = evall[evall$id ==1, ]

# All of the simulation data
simall = simres$simall
simall$id = as.factor(simall$id)

# Timecourse
psim =
  plot_sr_tc(
    sro    = simres,
    dvcols = "Cc")
psim$fig

# Events
pev =
  plot_sr_ev(
    sro    = simres,
    ylog   = FALSE)
pev$fig

}
```

`fetch_rxtc` *Extracts Timecourse and Merges Covariates*

Description

Takes the output of `rxSolve()` and merges in any missing covariates that are present in params but not in sim

Usage

```
fetch_rxtc(rx_details, sim)
```

Arguments

`rx_details`	Output of `fetch_rxinfo()`
`sim`	output of `rxSolve()`

Value

Dataframe of the simulated time course.

`MB_append_report` *Append Report Elements*

Description

Appends report elements to a formods report.

Usage

```
MB_append_report(state, rpt, rpttype, gen_code_only = FALSE)
```

Arguments

`state`	MB state from `MB_fetch_state()`
`rpt`	Report with the current content of the report which will be appended to in this function. For details on the structure see the documentation for `formods::FM_generate_report()`
`rpttype`	Type of report to generate (supported "xlsx", "pptx", "docx").
`gen_code_only`	Boolean value indicating that only code should be generated (`FALSE`).

Value

list containing the following elements

- isgood: Return status of the function.
- hasrptele: Boolean indicator if the module has any reportable elements.
- code: Code to generate reporting elements.
- msgs: Messages to be passed back to the user.
- rpt: Report with any additions passed back to the user.

See Also

`formods::FM_generate_report()`

`MB_build_code` *Build Code to Generate Model*

Description

Takes the function definition from an rxode object, a function object name and an rxode object name and creates the code to build those objects.

Usage

```
MB_build_code(
  state,
  session,
  fcn_def,
  time_scale,
  fcn_obj_name,
  rx_obj_name,
  ts_obj_name
)
```

Arguments

`state`	MB state from `MB_fetch_state()`
`session`	Shiny session variable
`fcn_def`	Character string containing the function definition for the model
`time_scale`	Short name for the model timescale (see names of stateMCformatting$time_scales$choices).
`fcn_obj_name`	Object name of the function to create.
`rx_obj_name`	Object name of the rxode2 object to create.
`ts_obj_name`	Object name of the tiemscale object to create.

Value

List with the following elements

- model_code Block of code to create the model in the context of a larger script.
- model_code_sa Same as the `model_code` element but meant to stand alone.

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
  state = MB_fetch_state(id              = "MB",
                       input           = input,
                       session         = session,
                       FM_yaml_file    = FM_yaml_file,
                       MOD_yaml_file   = MOD_yaml_file,
                       react_state     = NULL)

  # This will provide a list of the available models
  models = MB_fetch_catalog(state)
  # This is a summary of the tables in the model:
  models$summary

  # This will test the models in the catalog, set as_cran
  # to FALSE to test all the models.
  mtres = MB_test_catalog(state, as_cran=TRUE)
  mtres$isgood


  # Creates a new empty element
  state = MB_new_element(state)

  # Delete the current element
  state = MB_del_current_element(state)

  # Fetch a list of the current element
  element = MB_fetch_current_element(state)

  # This will attach a model to it:
  # Pulling the first model from the catalog
```

```
  fcn_def  = models[["summary"]][1, ][["Model"]]
  fcn_obj  = models[["summary"]][1, ][["Object"]]
  mdl_type = models[["summary"]][1, ][["Type"]]
  fcn_desc = models[["summary"]][1, ][["Description"]]

  # This will build the rxode2 object from the model
  mk_rx_res = mk_rx_obj(
    type  = mdl_type,
    model = list(fcn_def = fcn_def,
                 fcn_obj = fcn_obj))

  # This will attach the model to the current element
  element = MB_update_model(
    state       = state,
    session     = session,
    current_ele = element,
    rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
    note        = fcn_desc,
    reset       = TRUE)

  # You can now place element back in the state
  state = MB_set_current_element(state, element)

  # This will fetch the current component
  component = MB_fetch_component(state, element)


  fares = MB_fetch_appends(state, element)

  # You can use the component to build the code to generate the model:
  gen_code =
    MB_build_code(state        = state, session = session,
                  fcn_def      = component[["fcn_def"]],
                  time_scale   = element[["ui"]][["time_scale"]],
                  fcn_obj_name = "my_fcn_obj",
                  rx_obj_name  = "my_obj_name",
                  ts_obj_name  = "my_ts_name")

  # Model code to be included in a larger script
  message(paste0(gen_code$model_code, collapse="\n"))

  # Stand-alone model code
  message(paste0(gen_code$model_code_sa, collapse="\n"))

  # This will fetch the code to regenerate all of the components of this module
  message(MB_fetch_code(state))
}
```

`MB_del_current_element` *Deletes Current model*

Description

Takes a MB state and deletes the current model. If that is the last element, then a new default will be added.

Usage

```
MB_del_current_element(state)
```

Arguments

`state`	MB state from `MB_fetch_state()`

Value

MB state object with the current model deleted.

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
  state = MB_fetch_state(id              = "MB",
                         input           = input,
                         session         = session,
                         FM_yaml_file    = FM_yaml_file,
                         MOD_yaml_file   = MOD_yaml_file,
                         react_state     = NULL)

  # This will provide a list of the available models
  models = MB_fetch_catalog(state)
  # This is a summary of the tables in the model:
  models$summary

  # This will test the models in the catalog, set as_cran
  # to FALSE to test all the models.
  mtres = MB_test_catalog(state, as_cran=TRUE)
  mtres$isgood


  # Creates a new empty element
```

```
state = MB_new_element(state)

# Delete the current element
state = MB_del_current_element(state)

# Fetch a list of the current element
element = MB_fetch_current_element(state)

# This will attach a model to it:
# Pulling the first model from the catalog
fcn_def  = models[["summary"]][1, ][["Model"]]
fcn_obj  = models[["summary"]][1, ][["Object"]]
mdl_type = models[["summary"]][1, ][["Type"]]
fcn_desc = models[["summary"]][1, ][["Description"]]

# This will build the rxode2 object from the model
mk_rx_res = mk_rx_obj(
  type  = mdl_type,
  model = list(fcn_def = fcn_def,
               fcn_obj = fcn_obj))

# This will attach the model to the current element
element = MB_update_model(
  state       = state,
  session     = session,
  current_ele = element,
  rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
  note        = fcn_desc,
  reset       = TRUE)

# You can now place element back in the state
state = MB_set_current_element(state, element)

# This will fetch the current component
component = MB_fetch_component(state, element)


fares = MB_fetch_appends(state, element)

# You can use the component to build the code to generate the model:
gen_code =
  MB_build_code(state        = state, session = session,
                fcn_def      = component[["fcn_def"]],
                time_scale   = element[["ui"]][["time_scale"]],
                fcn_obj_name = "my_fcn_obj",
                rx_obj_name  = "my_obj_name",
                ts_obj_name  = "my_ts_name")

# Model code to be included in a larger script
message(paste0(gen_code$model_code, collapse="\n"))

# Stand-alone model code
message(paste0(gen_code$model_code_sa, collapse="\n"))
```

```
  # This will fetch the code to regenerate all of the components of this module
  message(MB_fetch_code(state))
}
```

`MB_fetch_appends` *Fetches List of Available Models*

Description

Creates a catalog of the models available in the system file.

Usage

```
MB_fetch_appends(state, current_ele)
```

Arguments

`state`	MB state from `MB_fetch_state()`
`current_ele`	MB model element from `MB_fetch_current_element()`

Value

List with the following attributes:

- isgood: Boolean variable indicating success or failure.
- msgs: Messages to be passed back to the user.
- hasappends: Boolean variable indicating if appendable models were found.
- select_plain: Flat list with the models (ungrouped).
- choicesOpt List witht he subtext filled out.

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
```

```
state = MB_fetch_state(id              = "MB",
                       input           = input,
                       session         = session,
                       FM_yaml_file    = FM_yaml_file,
                       MOD_yaml_file   = MOD_yaml_file,
                       react_state     = NULL)

# This will provide a list of the available models
models = MB_fetch_catalog(state)
# This is a summary of the tables in the model:
models$summary

# This will test the models in the catalog, set as_cran
# to FALSE to test all the models.
mtres = MB_test_catalog(state, as_cran=TRUE)
mtres$isgood


# Creates a new empty element
state = MB_new_element(state)

# Delete the current element
state = MB_del_current_element(state)

# Fetch a list of the current element
element = MB_fetch_current_element(state)

# This will attach a model to it:
# Pulling the first model from the catalog
fcn_def  = models[["summary"]][1, ][["Model"]]
fcn_obj  = models[["summary"]][1, ][["Object"]]
mdl_type = models[["summary"]][1, ][["Type"]]
fcn_desc = models[["summary"]][1, ][["Description"]]

# This will build the rxode2 object from the model
mk_rx_res = mk_rx_obj(
  type  = mdl_type,
  model = list(fcn_def = fcn_def,
               fcn_obj = fcn_obj))

# This will attach the model to the current element
element = MB_update_model(
  state       = state,
  session     = session,
  current_ele = element,
  rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
  note        = fcn_desc,
  reset       = TRUE)

# You can now place element back in the state
state = MB_set_current_element(state, element)

# This will fetch the current component
```

```
  component = MB_fetch_component(state, element)


  fares = MB_fetch_appends(state, element)

  # You can use the component to build the code to generate the model:
  gen_code =
    MB_build_code(state        = state, session = session,
                  fcn_def      = component[["fcn_def"]],
                  time_scale   = element[["ui"]][["time_scale"]],
                  fcn_obj_name = "my_fcn_obj",
                  rx_obj_name  = "my_obj_name",
                  ts_obj_name  = "my_ts_name")

  # Model code to be included in a larger script
  message(paste0(gen_code$model_code, collapse="\n"))

  # Stand-alone model code
  message(paste0(gen_code$model_code_sa, collapse="\n"))

  # This will fetch the code to regenerate all of the components of this module
  message(MB_fetch_code(state))
}
```

`MB_fetch_catalog` *Fetches List of Available Models*

Description

Creates a catalog of the models available in the system file.

Usage

```
MB_fetch_catalog(state)
```

Arguments

`state`	MB state from `MB_fetch_state()`

Value

List with the following attributes:

- summary: Dataframe with a summary of the models in the catlog
- sources: Same information a that found in the summary table but in list form.
- select_group: List with the models grouped by source.
- select_plain: Flat list with the models (ungrouped).
- select_subtext: Subtext for pulldown menus.

- msgs: Messages to be passed back to the user.
- hasmdl: Boolean value indicating if any models were found.
- isgood: Boolean variable indicating success or failure.

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
  state = MB_fetch_state(id              = "MB",
                         input           = input,
                         session         = session,
                         FM_yaml_file    = FM_yaml_file,
                         MOD_yaml_file   = MOD_yaml_file,
                         react_state     = NULL)

  # This will provide a list of the available models
  models = MB_fetch_catalog(state)
  # This is a summary of the tables in the model:
  models$summary

  # This will test the models in the catalog, set as_cran
  # to FALSE to test all the models.
  mtres = MB_test_catalog(state, as_cran=TRUE)
  mtres$isgood


  # Creates a new empty element
  state = MB_new_element(state)

  # Delete the current element
  state = MB_del_current_element(state)

  # Fetch a list of the current element
  element = MB_fetch_current_element(state)

  # This will attach a model to it:
  # Pulling the first model from the catalog
  fcn_def  = models[["summary"]][1, ][["Model"]]
  fcn_obj  = models[["summary"]][1, ][["Object"]]
```

```
  mdl_type = models[["summary"]][1, ][["Type"]]
  fcn_desc = models[["summary"]][1, ][["Description"]]

  # This will build the rxode2 object from the model
  mk_rx_res = mk_rx_obj(
    type  = mdl_type,
    model = list(fcn_def = fcn_def,
                 fcn_obj = fcn_obj))

  # This will attach the model to the current element
  element = MB_update_model(
    state       = state,
    session     = session,
    current_ele = element,
    rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
    note        = fcn_desc,
    reset       = TRUE)

  # You can now place element back in the state
  state = MB_set_current_element(state, element)

  # This will fetch the current component
  component = MB_fetch_component(state, element)


  fares = MB_fetch_appends(state, element)

  # You can use the component to build the code to generate the model:
  gen_code =
    MB_build_code(state        = state, session = session,
                  fcn_def      = component[["fcn_def"]],
                  time_scale   = element[["ui"]][["time_scale"]],
                  fcn_obj_name = "my_fcn_obj",
                  rx_obj_name  = "my_obj_name",
                  ts_obj_name  = "my_ts_name")

  # Model code to be included in a larger script
  message(paste0(gen_code$model_code, collapse="\n"))

  # Stand-alone model code
  message(paste0(gen_code$model_code_sa, collapse="\n"))

  # This will fetch the code to regenerate all of the components of this module
  message(MB_fetch_code(state))
}
```

`MB_fetch_code` *Fetch Module Code*

Description

Fetches the code to generate results seen in the app

Usage

```
MB_fetch_code(state)
```

Arguments

`state`	MB state from `MB_fetch_state()`

Value

Character object vector with the lines of code

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
  state = MB_fetch_state(id              = "MB",
                         input           = input,
                         session         = session,
                         FM_yaml_file    = FM_yaml_file,
                         MOD_yaml_file   = MOD_yaml_file,
                         react_state     = NULL)

  # This will provide a list of the available models
  models = MB_fetch_catalog(state)
  # This is a summary of the tables in the model:
  models$summary

  # This will test the models in the catalog, set as_cran
  # to FALSE to test all the models.
  mtres = MB_test_catalog(state, as_cran=TRUE)
  mtres$isgood


  # Creates a new empty element
  state = MB_new_element(state)

  # Delete the current element
  state = MB_del_current_element(state)
```

```
  # Fetch a list of the current element
  element = MB_fetch_current_element(state)

  # This will attach a model to it:
  # Pulling the first model from the catalog
  fcn_def  = models[["summary"]][1, ][["Model"]]
  fcn_obj  = models[["summary"]][1, ][["Object"]]
  mdl_type = models[["summary"]][1, ][["Type"]]
  fcn_desc = models[["summary"]][1, ][["Description"]]

  # This will build the rxode2 object from the model
  mk_rx_res = mk_rx_obj(
    type  = mdl_type,
    model = list(fcn_def = fcn_def,
                 fcn_obj = fcn_obj))

  # This will attach the model to the current element
  element = MB_update_model(
    state       = state,
    session     = session,
    current_ele = element,
    rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
    note        = fcn_desc,
    reset       = TRUE)

  # You can now place element back in the state
  state = MB_set_current_element(state, element)

  # This will fetch the current component
  component = MB_fetch_component(state, element)


  fares = MB_fetch_appends(state, element)

  # You can use the component to build the code to generate the model:
  gen_code =
    MB_build_code(state        = state, session = session,
                  fcn_def      = component[["fcn_def"]],
                  time_scale   = element[["ui"]][["time_scale"]],
                  fcn_obj_name = "my_fcn_obj",
                  rx_obj_name  = "my_obj_name",
                  ts_obj_name  = "my_ts_name")

  # Model code to be included in a larger script
  message(paste0(gen_code$model_code, collapse="\n"))

  # Stand-alone model code
  message(paste0(gen_code$model_code_sa, collapse="\n"))

  # This will fetch the code to regenerate all of the components of this module
  message(MB_fetch_code(state))
}
```

`MB_fetch_component` *Fetch Selected Current Model Component*

Description

Fetches the selected component of the provided model.

Usage

```
MB_fetch_component(state, current_ele, component_id = NULL)
```

Arguments

`state`	MB state from `MB_fetch_state()`
`current_ele`	MB model element from `MB_fetch_current_element()`
`component_id`	The numeric component id to select (default `NULL`) will return the selected ID.

Value

list with the current component with the following attributes

- isgood: Boolean object indicating success.
- rx_obj: rxode2 object for the model.
- ts_obj: timescale object for the model.
- fcn_def: Just the model function definition.
- note: Note field from the components_table
- model_code: Code to generate model.
- model_code_sa: Stand-alone code to generate model with
- msgs: Messages to be passed back to the user.

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")
```

```
# Creating an empty state object
state = MB_fetch_state(id              = "MB",
                       input           = input,
                       session         = session,
                       FM_yaml_file    = FM_yaml_file,
                       MOD_yaml_file   = MOD_yaml_file,
                       react_state     = NULL)

# This will provide a list of the available models
models = MB_fetch_catalog(state)
# This is a summary of the tables in the model:
models$summary

# This will test the models in the catalog, set as_cran
# to FALSE to test all the models.
mtres = MB_test_catalog(state, as_cran=TRUE)
mtres$isgood


# Creates a new empty element
state = MB_new_element(state)

# Delete the current element
state = MB_del_current_element(state)

# Fetch a list of the current element
element = MB_fetch_current_element(state)

# This will attach a model to it:
# Pulling the first model from the catalog
fcn_def  = models[["summary"]][1, ][["Model"]]
fcn_obj  = models[["summary"]][1, ][["Object"]]
mdl_type = models[["summary"]][1, ][["Type"]]
fcn_desc = models[["summary"]][1, ][["Description"]]

# This will build the rxode2 object from the model
mk_rx_res = mk_rx_obj(
  type  = mdl_type,
  model = list(fcn_def = fcn_def,
               fcn_obj = fcn_obj))

# This will attach the model to the current element
element = MB_update_model(
  state       = state,
  session     = session,
  current_ele = element,
  rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
  note        = fcn_desc,
  reset       = TRUE)

# You can now place element back in the state
state = MB_set_current_element(state, element)
```

```
    # This will fetch the current component
    component = MB_fetch_component(state, element)


    fares = MB_fetch_appends(state, element)

    # You can use the component to build the code to generate the model:
    gen_code =
      MB_build_code(state        = state, session = session,
                    fcn_def       = component[["fcn_def"]],
                    time_scale    = element[["ui"]][["time_scale"]],
                    fcn_obj_name  = "my_fcn_obj",
                    rx_obj_name   = "my_obj_name",
                    ts_obj_name   = "my_ts_name")

    # Model code to be included in a larger script
    message(paste0(gen_code$model_code, collapse="\n"))

    # Stand-alone model code
    message(paste0(gen_code$model_code_sa, collapse="\n"))

    # This will fetch the code to regenerate all of the components of this module
    message(MB_fetch_code(state))
  }
```

`MB_fetch_current_element` *Fetches Current model*

Description

Takes a MB state and returns the current active model object.

Usage

```
MB_fetch_current_element(state)
```

Arguments

`state`	MB state from `MB_fetch_state()`

Value

List containing the details of the active data view. The structure of this list is the same as the structure of `state$MB$elements` in the output of `MB_fetch_state()`.

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
  state = MB_fetch_state(id              = "MB",
                         input           = input,
                         session         = session,
                         FM_yaml_file    = FM_yaml_file,
                         MOD_yaml_file   = MOD_yaml_file,
                         react_state     = NULL)

  # This will provide a list of the available models
  models = MB_fetch_catalog(state)
  # This is a summary of the tables in the model:
  models$summary

  # This will test the models in the catalog, set as_cran
  # to FALSE to test all the models.
  mtres = MB_test_catalog(state, as_cran=TRUE)
  mtres$isgood


  # Creates a new empty element
  state = MB_new_element(state)

  # Delete the current element
  state = MB_del_current_element(state)

  # Fetch a list of the current element
  element = MB_fetch_current_element(state)

  # This will attach a model to it:
  # Pulling the first model from the catalog
  fcn_def  = models[["summary"]][1, ][["Model"]]
  fcn_obj  = models[["summary"]][1, ][["Object"]]
  mdl_type = models[["summary"]][1, ][["Type"]]
  fcn_desc = models[["summary"]][1, ][["Description"]]

  # This will build the rxode2 object from the model
  mk_rx_res = mk_rx_obj(
```

```
    type  = mdl_type,
    model = list(fcn_def = fcn_def,
                 fcn_obj = fcn_obj))

  # This will attach the model to the current element
  element = MB_update_model(
    state       = state,
    session     = session,
    current_ele = element,
    rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
    note        = fcn_desc,
    reset       = TRUE)

  # You can now place element back in the state
  state = MB_set_current_element(state, element)

  # This will fetch the current component
  component = MB_fetch_component(state, element)


  fares = MB_fetch_appends(state, element)

  # You can use the component to build the code to generate the model:
  gen_code =
    MB_build_code(state        = state, session = session,
                  fcn_def        = component[["fcn_def"]],
                  time_scale     = element[["ui"]][["time_scale"]],
                  fcn_obj_name   = "my_fcn_obj",
                  rx_obj_name    = "my_obj_name",
                  ts_obj_name    = "my_ts_name")

  # Model code to be included in a larger script
  message(paste0(gen_code$model_code, collapse="\n"))

  # Stand-alone model code
  message(paste0(gen_code$model_code_sa, collapse="\n"))

  # This will fetch the code to regenerate all of the components of this module
  message(MB_fetch_code(state))
}
```

`MB_fetch_mdl` *Fetch Model Builder Module Models*

Description

Fetches the models contained in the module.

Usage

```
MB_fetch_mdl(state)
```

Arguments

`state`	MB state from `MB_fetch_state()`

Value

list containing the following elements

- isgood: Return status of the function.
- hasmdl: Boolean indicator if the module has any models
- msgs: Messages to be passed back to the user.
- mdl: List with models. Each list element has the name of the R-object for that dataset. Each element has the following structure:
  - label: Text label for the model (e.g. one-compartment model).
  - MOD_TYPE: Type of module.
  - id: Module ID.
  - rx_obj: The rxode2 object.
  - rx_obj_name: The rxode2 object name that holds the model.
  - ts_obj List with elements system and details
  - ts_obj_name: The object name that holds the model time scale information.
  - fcn_def: Text to define the model
  - MDLMETA: Notes about the model.
  - code: Code to generate the model.
  - checksum: Module checksum.
  - MDLchecksum: Model checksum.

Examples

```
# We need a module state:
sess_res = MB_test_mksession()
state = sess_res$state

mdls = MB_fetch_mdl(state)

names(mdls)
```

`MB_fetch_state` *Fetch Model Builder State*

Description

Merges default app options with the changes made in the UI

Usage

```
MB_fetch_state(
  id,
  id_ASM,
  input,
  session,
  FM_yaml_file,
  MOD_yaml_file,
  react_state
)
```

Arguments

`id`	Shiny module ID
`id_ASM`	ID string for the app state management module used to save and load app states
`input`	Shiny input variable
`session`	Shiny session variable
`FM_yaml_file`	App configuration file with FM as main section.
`MOD_yaml_file`	Module configuration file with MC as main section.
`react_state`	Variable passed to server to allow reaction outside of module (`NULL`)

Value

list containing the current state of the app including default values from the yaml file as well as any changes made by the user. The list has the following structure:

- yaml: Full contents of the supplied yaml file.
- MC: Module components of the yaml file.
- MB:
  - isgood: Boolean object indicating if the file was successfully loaded.
  - checksum: This is an MD5 sum of the contents element and can be used to detect changes in the state.
- MOD_TYPE: Character data containing the type of module `"MB"`
- id: Character data containing the module id module in the session variable.
- FM_yaml_file: App configuration file with FM as main section.
- MOD_yaml_file: Module configuration file with MC as main section.

Examples

```
# Within shiny both session and input variables will exist,
# this creates examples here for testing purposes:
sess_res = MB_test_mksession()
session = sess_res$session
input   = sess_res$input
```

```
# Configuration files
FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

# Creating an empty state object
state = MB_fetch_state(id              = "MB",
                       id_ASM          = "ASM",
                       input           = input,
                       session         = session,
                       FM_yaml_file    = FM_yaml_file,
                       MOD_yaml_file   = MOD_yaml_file,
                       react_state     = NULL)
```

`MB_init_state` *Initialize MB Module State*

Description

Creates a list of the initialized module state

Usage

```
MB_init_state(FM_yaml_file, MOD_yaml_file, id, session)
```

Arguments

`FM_yaml_file`	App configuration file with FM as main section.
`MOD_yaml_file`	Module configuration file with MC as main section.
`id`	ID string for the module.
`session`	Shiny session variable

Value

list containing an empty MB state

Examples

```
# Within shiny both session and input variables will exist,
# this creates examples here for testing purposes:
sess_res = MB_test_mksession()
session = sess_res$session
input   = sess_res$input

state = MB_init_state(
   FM_yaml_file  = system.file(package = "formods",
                               "templates",
                               "formods.yaml"),
   MOD_yaml_file = system.file(package = "ruminate",
                               "templates",
```

```
                                    "MB.yaml"),
   id              = "MB",
   session         = session)

state
```

`MB_mk_preload` *Make List of Current MB State*

Description

Reads in the app state from yaml files.

Usage

```
MB_mk_preload(state)
```

Arguments

`state`	MB state object

Value

list with the following elements

- isgood: Boolean indicating the exit status of the function.
- msgs: Messages to be passed back to the user.
- yaml_list: Lists with preload components.

Examples

```
sess_res = MB_test_mksession()
state = sess_res$state
res = MB_mk_preload(state)
```

`MB_new_element` *New Model Building Model*

Description

Appends a new empty model to the MB state object and makes this new model the active model.

Usage

```
MB_new_element(state)
```

Arguments

`state`	MB state from `MB_fetch_state()`

Value

MB state object containing a new model and that model is set as the current active model. See the help for `MB_fetch_state()` for model format.

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
  state = MB_fetch_state(id              = "MB",
                       input           = input,
                       session         = session,
                       FM_yaml_file    = FM_yaml_file,
                       MOD_yaml_file   = MOD_yaml_file,
                       react_state     = NULL)

  # This will provide a list of the available models
  models = MB_fetch_catalog(state)
  # This is a summary of the tables in the model:
  models$summary

  # This will test the models in the catalog, set as_cran
  # to FALSE to test all the models.
  mtres = MB_test_catalog(state, as_cran=TRUE)
  mtres$isgood


  # Creates a new empty element
  state = MB_new_element(state)

  # Delete the current element
  state = MB_del_current_element(state)

  # Fetch a list of the current element
  element = MB_fetch_current_element(state)
```

```
  # This will attach a model to it:
  # Pulling the first model from the catalog
  fcn_def  = models[["summary"]][1, ][["Model"]]
  fcn_obj  = models[["summary"]][1, ][["Object"]]
  mdl_type = models[["summary"]][1, ][["Type"]]
  fcn_desc = models[["summary"]][1, ][["Description"]]

  # This will build the rxode2 object from the model
  mk_rx_res = mk_rx_obj(
    type  = mdl_type,
    model = list(fcn_def = fcn_def,
                 fcn_obj = fcn_obj))

  # This will attach the model to the current element
  element = MB_update_model(
    state       = state,
    session     = session,
    current_ele = element,
    rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
    note        = fcn_desc,
    reset       = TRUE)

  # You can now place element back in the state
  state = MB_set_current_element(state, element)

  # This will fetch the current component
  component = MB_fetch_component(state, element)


  fares = MB_fetch_appends(state, element)

  # You can use the component to build the code to generate the model:
  gen_code =
    MB_build_code(state        = state, session = session,
                  fcn_def      = component[["fcn_def"]],
                  time_scale   = element[["ui"]][["time_scale"]],
                  fcn_obj_name = "my_fcn_obj",
                  rx_obj_name  = "my_obj_name",
                  ts_obj_name  = "my_ts_name")

  # Model code to be included in a larger script
  message(paste0(gen_code$model_code, collapse="\n"))

  # Stand-alone model code
  message(paste0(gen_code$model_code_sa, collapse="\n"))

  # This will fetch the code to regenerate all of the components of this module
  message(MB_fetch_code(state))
}
```

`MB_preload` *Preload Data for MB Module*

Description

Populates the supplied session variable with information from list of sources.

Usage

```
MB_preload(
  session,
  src_list,
  yaml_res,
  mod_ID = NULL,
  react_state = list(),
  quickload = FALSE
)
```

Arguments

`session`	Shiny session variable (in app) or a list (outside of app)
`src_list`	List of preload data (all read together with module IDs at the top level)
`yaml_res`	List data from module yaml config
`mod_ID`	Module ID of the module being loaded.
`react_state`	Reactive shiny object (in app) or a list (outside of app) used to trigger reactions.
`quickload`	Logical `TRUE` to load reduced analysis `FALSE` to load the full analysis

Value

list with the following elements

- isgood: Boolean indicating the exit status of the function.
- msgs: Messages to be passed back to the user.
- session: Session object
- input: The value of the shiny input at the end of the session initialization.
- state: App state.
- react_state: The `react_state` components.

`MB_Server` *Model Builder State Server*

Description

Server function for the Model Builder Shiny Module

Usage

```
MB_Server(
  id,
  id_ASM = "ASM",
  FM_yaml_file = system.file(package = "formods", "templates", "formods.yaml"),
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml"),
  deployed = FALSE,
  react_state = NULL
)
```

Arguments

`id`	An ID string that corresponds with the ID used to call the modules UI elements
`id_ASM`	ID string for the app state managment module used to save and load app states
`FM_yaml_file`	App configuration file with FM as main section.
`MOD_yaml_file`	Module configuration file with MC as main section.
`deployed`	Boolean variable indicating whether the app is deployed or not.
`react_state`	Variable passed to server to allow reaction outside of module (`NULL`)

Value

MB Server object

`MB_set_current_element`

Sets the Value for the Current model

Description

Takes a MB state and returns the current active model

Usage

```
MB_set_current_element(state, element)
```

Arguments

`state`	MB state from `MB_fetch_state()`
`element`	Element list from `MB_fetch_current_element()`

Value

MB state object with the current model set using the supplied value.

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
  state = MB_fetch_state(id              = "MB",
                         input           = input,
                         session         = session,
                         FM_yaml_file    = FM_yaml_file,
                         MOD_yaml_file   = MOD_yaml_file,
                         react_state     = NULL)

  # This will provide a list of the available models
  models = MB_fetch_catalog(state)
  # This is a summary of the tables in the model:
  models$summary

  # This will test the models in the catalog, set as_cran
  # to FALSE to test all the models.
  mtres = MB_test_catalog(state, as_cran=TRUE)
  mtres$isgood


  # Creates a new empty element
  state = MB_new_element(state)

  # Delete the current element
  state = MB_del_current_element(state)

  # Fetch a list of the current element
  element = MB_fetch_current_element(state)

  # This will attach a model to it:
  # Pulling the first model from the catalog
  fcn_def  = models[["summary"]][1, ][["Model"]]
  fcn_obj  = models[["summary"]][1, ][["Object"]]
  mdl_type = models[["summary"]][1, ][["Type"]]
  fcn_desc = models[["summary"]][1, ][["Description"]]

  # This will build the rxode2 object from the model
  mk_rx_res = mk_rx_obj(
```

```
    type  = mdl_type,
    model = list(fcn_def = fcn_def,
                 fcn_obj = fcn_obj))

  # This will attach the model to the current element
  element = MB_update_model(
    state       = state,
    session     = session,
    current_ele = element,
    rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
    note        = fcn_desc,
    reset       = TRUE)

  # You can now place element back in the state
  state = MB_set_current_element(state, element)

  # This will fetch the current component
  component = MB_fetch_component(state, element)


  fares = MB_fetch_appends(state, element)

  # You can use the component to build the code to generate the model:
  gen_code =
    MB_build_code(state        = state, session = session,
                  fcn_def        = component[["fcn_def"]],
                  time_scale     = element[["ui"]][["time_scale"]],
                  fcn_obj_name   = "my_fcn_obj",
                  rx_obj_name    = "my_obj_name",
                  ts_obj_name    = "my_ts_name")

  # Model code to be included in a larger script
  message(paste0(gen_code$model_code, collapse="\n"))

  # Stand-alone model code
  message(paste0(gen_code$model_code_sa, collapse="\n"))

  # This will fetch the code to regenerate all of the components of this module
  message(MB_fetch_code(state))
}
```

`MB_test_catalog` *Tests the Model Catalog*

Description

Reads in models in the catalog and attempts to build them.

Usage

```
MB_test_catalog(state, as_cran = FALSE, verbose = TRUE)
```

Arguments

`state`	MB state from `MB_fetch_state()`
`as_cran`	Boolean to indicate if you're running this on CRAN
`verbose`	Boolean to indicate if messages should be displayed.

Value

List with the following attributes:

- isgood: Boolean varaible indicating if all the models in the catalog passed the test.
- msgs: Messages indicating if the test was successful or not.

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
  state = MB_fetch_state(id              = "MB",
                         input           = input,
                         session         = session,
                         FM_yaml_file    = FM_yaml_file,
                         MOD_yaml_file   = MOD_yaml_file,
                         react_state     = NULL)

  # This will provide a list of the available models
  models = MB_fetch_catalog(state)
  # This is a summary of the tables in the model:
  models$summary

  # This will test the models in the catalog, set as_cran
  # to FALSE to test all the models.
  mtres = MB_test_catalog(state, as_cran=TRUE)
  mtres$isgood


  # Creates a new empty element
  state = MB_new_element(state)

  # Delete the current element
```

```
state = MB_del_current_element(state)

# Fetch a list of the current element
element = MB_fetch_current_element(state)

# This will attach a model to it:
# Pulling the first model from the catalog
fcn_def  = models[["summary"]][1, ][["Model"]]
fcn_obj  = models[["summary"]][1, ][["Object"]]
mdl_type = models[["summary"]][1, ][["Type"]]
fcn_desc = models[["summary"]][1, ][["Description"]]

# This will build the rxode2 object from the model
mk_rx_res = mk_rx_obj(
  type  = mdl_type,
  model = list(fcn_def = fcn_def,
               fcn_obj = fcn_obj))

# This will attach the model to the current element
element = MB_update_model(
  state       = state,
  session     = session,
  current_ele = element,
  rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
  note        = fcn_desc,
  reset       = TRUE)

# You can now place element back in the state
state = MB_set_current_element(state, element)

# This will fetch the current component
component = MB_fetch_component(state, element)


fares = MB_fetch_appends(state, element)

# You can use the component to build the code to generate the model:
gen_code =
  MB_build_code(state        = state, session = session,
              fcn_def        = component[["fcn_def"]],
              time_scale     = element[["ui"]][["time_scale"]],
              fcn_obj_name   = "my_fcn_obj",
              rx_obj_name    = "my_obj_name",
              ts_obj_name    = "my_ts_name")

# Model code to be included in a larger script
message(paste0(gen_code$model_code, collapse="\n"))

# Stand-alone model code
message(paste0(gen_code$model_code_sa, collapse="\n"))

# This will fetch the code to regenerate all of the components of this module
message(MB_fetch_code(state))
```

```
}
```

`MB_test_mksession` *Populate Session Data for Module Testing*

Description

Populates the supplied session variable for testing.

Usage

```
MB_test_mksession(session = list())
```

Arguments

`session`	Shiny session variable (in app) or a list (outside of app)

Value

The MB portion of the `all_sess_res` returned from `FM_app_preload`

See Also

`FM_app_preload`

Examples

```
session = shiny::MockShinySession$new()
sess_res = MB_test_mksession(session=session)
```

`MB_update_checksum` *Update MB Module Checksum*

Description

Takes a MB state and updates the checksum used to trigger downstream updates

Usage

```
MB_update_checksum(state)
```

Arguments

`state`	MB state from `MB_fetch_state()`

Value

MB state object with the checksum updated

Examples

```
# Within shiny both session and input variables will exist,
# this creates examples here for testing purposes:
sess_res = MB_test_mksession()
session = sess_res$session
input   = sess_res$input

# We also need a state variable
state = sess_res$state

state = MB_update_checksum(state)
```

`MB_update_model` *Updates Current Element with rxode2 Model*

Description

Takes an rxode2 object and updates the model components of the current element.

Usage

```
MB_update_model(state, session, current_ele, rx_obj, note, reset = FALSE)
```

Arguments

`state`	MB state from `MB_fetch_state()`
`session`	Shiny session variable
`current_ele`	MB model element from `MB_fetch_current_element()`
`rx_obj`	rxode2 model from `rxode2::rxode2()`
`note`	text indicating what this update does (e.g. "added parameter")
`reset`	boolean indicating that the element needs to be reset (i.e. if you change the base model) default: `FALSE`.

Value

current_element with model attached

Examples

```
#library(ruminate)
# This will get the full session:
library(formods)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
  sess_res = MB_test_mksession()
  # This is just for CRAN
  #sess_res = MB_test_mksession(session=list())
  session = sess_res$session
  input   = sess_res$input

  # Configuration files
  FM_yaml_file  = system.file(package = "formods", "templates", "formods.yaml")
  MOD_yaml_file = system.file(package = "ruminate", "templates", "MB.yaml")

  # Creating an empty state object
  state = MB_fetch_state(id              = "MB",
                         input           = input,
                         session         = session,
                         FM_yaml_file    = FM_yaml_file,
                         MOD_yaml_file   = MOD_yaml_file,
                         react_state     = NULL)

  # This will provide a list of the available models
  models = MB_fetch_catalog(state)
  # This is a summary of the tables in the model:
  models$summary

  # This will test the models in the catalog, set as_cran
  # to FALSE to test all the models.
  mtres = MB_test_catalog(state, as_cran=TRUE)
  mtres$isgood


  # Creates a new empty element
  state = MB_new_element(state)

  # Delete the current element
  state = MB_del_current_element(state)

  # Fetch a list of the current element
  element = MB_fetch_current_element(state)

  # This will attach a model to it:
  # Pulling the first model from the catalog
  fcn_def  = models[["summary"]][1, ][["Model"]]
  fcn_obj  = models[["summary"]][1, ][["Object"]]
  mdl_type = models[["summary"]][1, ][["Type"]]
  fcn_desc = models[["summary"]][1, ][["Description"]]

  # This will build the rxode2 object from the model
  mk_rx_res = mk_rx_obj(
```

```
    type  = mdl_type,
    model = list(fcn_def = fcn_def,
                 fcn_obj = fcn_obj))

  # This will attach the model to the current element
  element = MB_update_model(
    state       = state,
    session     = session,
    current_ele = element,
    rx_obj      = mk_rx_res[["capture"]][["rx_obj"]],
    note        = fcn_desc,
    reset       = TRUE)

  # You can now place element back in the state
  state = MB_set_current_element(state, element)

  # This will fetch the current component
  component = MB_fetch_component(state, element)


  fares = MB_fetch_appends(state, element)

  # You can use the component to build the code to generate the model:
  gen_code =
    MB_build_code(state        = state, session = session,
                  fcn_def        = component[["fcn_def"]],
                  time_scale     = element[["ui"]][["time_scale"]],
                  fcn_obj_name   = "my_fcn_obj",
                  rx_obj_name    = "my_obj_name",
                  ts_obj_name    = "my_ts_name")

  # Model code to be included in a larger script
  message(paste0(gen_code$model_code, collapse="\n"))

  # Stand-alone model code
  message(paste0(gen_code$model_code_sa, collapse="\n"))

  # This will fetch the code to regenerate all of the components of this module
  message(MB_fetch_code(state))
}
```

`mk_figure_ind_obs` *Creates Figures of Individual Observations from PKNCA Result*

Description

Takes the output of PKNCA and creates `ggplot` figures faceted by subject id highlighting of certain NCA aspects (e.g. points used for half-life)

Usage

```
mk_figure_ind_obs(
  nca_res,
  OBS_LAB = "Concentration ===CONCUNITS===",
  TIME_LAB = "Time ===TIMEUNITS===",
  OBS_STRING = "Observation",
  BLQ_STRING = "BLQ",
  NA_STRING = "Missing",
  log_scale = TRUE,
  scales = "fixed",
  nfrows = 4,
  nfcols = 3
)
```

Arguments

`nca_res`	Output of PKNCA.
`OBS_LAB`	Label of the observation axis with optional ===CONCUNITS=== placeholder for units.
`TIME_LAB`	Label of the time axis with optional ===TIMEUNITS=== placeholder for units.
`OBS_STRING`	Label for observation data.
`BLQ_STRING`	Label for BLQ data.
`NA_STRING`	Label for missing data.
`log_scale`	Boolean variable to control y-scale (`TRUE`: Log 10, `FALSE`: linear).
`scales`	String to determine the scales used when faceting. Can be either `"fixed"`, `"free"`, `"free_x"`, or `"free_y"`.
`nfrows`	Number of facet rows per page.
`nfcols`	Number of facet cols per page.

Value

List containing the element `figures` which is a list of figure pages (`"Figure 1"`, `"Figure 2"`, etc.). Each of these is a also a list containing two elements:

- gg: A ggplot object for that page.
- notes: Placeholder for future notes, but NULL now.

Examples

```
# We need a state variable to be define
sess_res = NCA_test_mksession()

state = sess_res$state

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)
```

```
# This is the raw PKNCA output
pknca_res = NCA_fetch_ana_pknca(state, current_ana)

# Building the figure
mk_res = mk_figure_ind_obs(nca_res = pknca_res)
mk_res$figures$Figure_1$gg
```

`mk_rx_obj` *Makes an rxode2 Object*

Description

Creates an rxode2 object from a model (either rxode2 function or a NONMEM file)

Usage

```
mk_rx_obj(type, model)
```

Arguments

`type`	Type of supplied model can be "rxode2", "NONMEM"
`model`	List containing the relevant information about the model. This will depend on the model types. • rxode2: The supplied model is in the rxode2 format. – fcn_def: Character string containing function definition. – fcn_obj: Name of the funciton object created in `fcn_def`. • NONMEM: The supplied model is in NONMEM format (either a control – model_file: Character string containing the NONMEM model file.

Value

Results of `FM_tc()` when running the model. This will include a field `isgood` which is a boolean variable indicating success or failure. See the documentation for `FM_tc()` for the format returned when evaluation results in a failure and how to address those. When successful the `capture` field will contain the following:

- fcn_obj: The function name.
- rx_obj: The built rxode2 object.

Examples

```
fcn_def = ' my_func = function ()
   {
       description <- "One compartment PK model with linear clearance"
       ini({
           lka <- 0.45
```

```
            label("Absorption rate (Ka)")
            lcl <- 1
            label("Clearance (CL)")
            lvc <- 3.45
            label("Central volume of distribution (V)")
            propSd <- c(0, 0.5)
            label("Proportional residual error (fraction)")
        })
        model({
            ka <- exp(lka)
            cl <- exp(lcl)
            vc <- exp(lvc)
            cp <- linCmt()
            cp ~ prop(propSd)
        })

   }'
fcn_obj = "my_func"
model = list(fcn_def = fcn_def,
             fcn_obj = fcn_obj)


rx_res = mk_rx_obj("rxode2", model)

# function object
rx_res[["capture"]][["fcn_obj"]]

# rxode2 object
rx_res[["capture"]][["rx_obj"]]
```

`mk_subjects` *Fetches Information from an rxode2 Object*

Description

This will provide information like parameter names, covriates, etc from an rxode2 object.

Usage

```
mk_subjects(object, nsub = 10, covs = NULL)
```

Arguments

`object`	rxode2 model object An ID string that corresponds with the ID used to call the modules UI elements.
`nsub`	Number of subjects to generate. If set to 1 it will return the typical values (IIV set to zero).
`covs`	List describing how covariates should be generated.

Details

See below.

The underlying simulations are run using `rxode2`, and as such we need an `rxode2` system object. From that we can either simulate subjects or load them from a file. Next we need to define a set of rules. These will be a set of conditions and actions. At each evaluation time point the conditions are evaluated. When a condition is met the actions associated with that condition are executed. For example, if during a visit (an evaluation time point) the trough PK is below a certain level (condition) we may want to increase the dosing regimen for the next dosing cycle (action).

Creating subjects:

Subjects are expected in a data frame with the following column headers:

- `id` Individual subject id
- Names of parameters and iiv as specified in the `ini` section of the `rxode2` function specification
- Names of covariates used in the model.

`mk_subjects()` — Creates subjects for simulation by sampling based on between-subject variability and generating covariate information based on user specifications.

Covariates:

The `covs` input is a list with the following structure:

- type: Can be either “fixed”, “discrete”, or “continuous”.
- sampling: This field is only needed for a “continuous” covariate ’ type and can be either “random”, “normal” or “log-normal”.
- values: This field depends on the type and optional sampling above.
  - fixed: A single value.
  - discrete: A vector of possible discrete elements.
  - continuous, random: Two values the first is the lower bound and the second is the upper bound.
  - continuous, normal: Two values the first is the mean and the second is the variance.
  - continuous, log-normal: Two values the first is the mean and the second is the variance.

This examples shows the `SEX_ID` randomly sampled from the values specified, `SUBTYPE_ID` fixed at a value, and `WT` sampled from a log-normal distribution.

```
covs = list(
  SEX_ID     = list(type     = "discrete",
                    values   = c(0,1)),
  SUBTYPE_ID = list(type     = "fixed",
                    values   = c(0)),
  WT         = list(type     = "continuous",
                    sampling = "log-normal",
                    values   = c(70, .15))
)
```

Rule-based simulations:

`simulate_rules()` — This will run simulations based on the rule definitions below.

Rules:

Rules are a named list where the list name can be a short descriptive label used to remember what the rule does. These names will be returned as columns in the simulated data frame.

- `condition`: Character string that evaluates to either `TRUE` or `FALSE`. When true the `action` portion will be triggered. For a list of objects available see the Rule-evaluation environment below.
- `fail_flag`: Flag set in the `rule_id` column when the condition is not met (set to `"false"` if not specified).
- `true_flag`: Flag set in the `rule_id` column when the condition is met (set to `"true"` if not specified).
- `action`: This is what the rule will trigger can be any of the following:
  - `type`: This defines the action type and can be either `"dose"`, `"set state"`, or `"manual"`.

Based on the `type` the `action` field will expect different elements.

Dosing:

- `action`
  - `type`: `"dose"`
  - `values`: Character string that evaluates as a numeric vector dosing amounts (e.g. `"c(3, 3, 3, 3)"`)
  - `times`: Character string that evaluates as a numeric vector of times (e.g. `"c(0, 14, 28, 42)"`)
  - `durations`: Character string that evaluates as a numeric vector of durations (e.g. `"c(0, 0, 0, 0)"`, zero for bolus dosing)

Changing a state value:

- `action`
  - `type`: `"set state"`
  - `state`: Character string with the name of the state to set (`"Ac"`)
  - `value`: Character string that evaluates as a numeric value for state (e.g. `"Ac/2"` would set the state to half the value of Ac at the evaluation point)

Manual modification of the simulation:

- `action`
  - `type`: `"manual"`
  - `code`: Character string of code to evaluate.

Rule-evaluation environment:

Beyond simple simulations it will be necessary to execute actions based on the current or previous state of the system. For this reason, when a `condition` or elements of the action (e.g., the `values`, `times` and `durations` of a dose action type) are being evaluated, the following objects will be available at each evaluation point:

- outputs: The value of each model output.
- states: The value of each named state or compartment.
- covariates: The value of each named covariate.
- subject-level parameters: The value of each named parameter.
- rule value: The last value the rule evaluated as.
- `id`: Current subject id.
- `time`: Current evaluation time.
- `SI_SUB_HISTORY`: A data frame of the simulation history of the current subject up to the current evaluation point.
- `SI_subjects`: The subjects data frame.

- `SI_eval_times`: Vector of the evaluation times.
- `SI_interval_ev`: The events table in it's current state for the given simulation interval.
- `SI_ev_history`: This is the history of the event table containing all the events leading up to the current interval.
- `SI_ud_history`: This is a free form object the user can define or alter within the "manual"action type (ud-user defined, history).

The following functions will be available::

- `SI_fpd`: This function will fetch the previous dose (fpd) for the given id and state. For example for the current `id` and the state `Ac` you would do the following:

```
SI_fpd(id=id, state="Ac")
```

Time scales:
You can include columns in your output for different time scales if you wish. You need to create a list in the format below. One element should be `system` with a short name for the system time scale. The next should be `details` which is a list containing short names for each time scale you want to include. Each of these is a list with a verbose name for the time scale (`verb`) and a numerical conversion indicating how that time scale relates to the others. Here we define weeks and days on the basis of seconds.

```
time_scales = list(system="days",
                details= list(
                  weeks = list(verb="Weeks",    conv=1/(60*60*24*7)),
                  days  = list(verb="Days",     conv=1/(60*60*24))))
```

Value

List with the following elements.

- isgood: Return status of the function.
- msgs: Error or warning messages if any issues were encountered.
- subjects: Data frame of parameters and covariates for the subjects generated.
- iCov: Data frame of the covariates.
- params: Data frame of the parameters.

See Also

`vignette("clinical_trial_simulation", package = "ruminate")`

Examples

```
library(formods)
library(ggplot2)

# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html

# None of this will work if rxode2 isn't installed:
if(is_installed("rxode2")){
library(rxode2)
```

```
set.seed(8675309)
rxSetSeed(8675309)

my_model = function ()
{
   description <- "One compartment PK model with linear clearance using differential equations"
    ini({
        lka <- 0.45
        label("Absorption rate (Ka)")
        lcl <- 1
        label("Clearance (CL)")
        lvc <- 3.45
        label("Central volume of distribution (V)")
        propSd <- c(0, 0.5)
        label("Proportional residual error (fraction)")
        etalcl ~ 0.1
    })
    model({
        ka <- exp(lka)
        cl <- exp(lcl + etalcl)
        vc <- exp(lvc)
        kel <- cl/vc
        d/dt(depot) <- -ka * depot
        d/dt(central) <- ka * depot - kel * central
        Cc <- central/vc
        Cc ~ prop(propSd)
    })
}

# This creates an rxode2 object
object  = rxode(my_model)

# If you want details about the parameters, states, etc
# in the model you can use this:
rxdetails = fetch_rxinfo(object)

rxdetails$elements

# Next we will create subjects. To do that we need to
# specify information about covariates:
nsub = 2
covs = list(
  WT         = list(type     = "continuous",
                    sampling = "log-normal",
                    values   = c(70, .15))
)

subs = mk_subjects(object = object,
                   nsub   = nsub,
                   covs   = covs)

head(subs$subjects)
```

```
rules = list(
  dose = list(
    condition = "TRUE",
    action    = list(
      type  = "dose",
      state     = "central",
      values    = "c(1)",
      times     = "c(0)",
      durations = "c(0)")
    )
)

# We evaulate the rules for dosing at time 0
eval_times =  0

# Stop 2 months after the last dose
output_times = seq(0, 56, 1)

# This runs the rule-based simulations
simres =
  simulate_rules(
    object        = object,
    subjects      = subs[["subjects"]],
    eval_times    = eval_times,
    output_times  = output_times,
    rules         = rules)

# First subject data:
sub_1 = simres$simall[simres$simall$id == 1, ]

# First subjects events
evall = as.data.frame(simres$evall)
ev_sub_1 = evall[evall$id ==1, ]

# All of the simulation data
simall = simres$simall
simall$id = as.factor(simall$id)

# Timecourse
psim =
  plot_sr_tc(
    sro    = simres,
    dvcols = "Cc")
psim$fig

# Events
pev =
  plot_sr_ev(
    sro    = simres,
    ylog   = FALSE)
pev$fig

}
```

`mk_table_ind_obs` *Creates Tables of Individual Observations from PKNCA Result*

Description

Takes the output of PKNCA and creates a tabular view of the individual observation data. This can be spread out of over several tables (pages) if necessary.

Usage

```
mk_table_ind_obs(
  nca_res,
  obnd = NULL,
  not_sampled = "NS",
  blq = "BLQ",
  digits = 3,
  text_format = "text",
  max_height = 7,
  max_width = 6.5,
  max_row = NULL,
  max_col = 9,
  notes_detect = NULL,
  rows_by = "time"
)
```

Arguments

`nca_res`	Output of PKNCA.
`obnd`	onbrand reporting object.
`not_sampled`	Character string to use for missing data when pivoting.
`blq`	Character string to use to indicate data below the level of quantification (value of 0 in the dataset).
`digits`	Number of significant figures to report (set to `NULL` to disable rounding)
`text_format`	Either `"md"` for markdown or `"text"` (default) for plain text.
`max_height`	Maximum height of the final table in inches (A value of `NULL` will use 100 inches).
`max_width`	Maximum width of the final table in inches (A value of `NULL` will use 100 inches).
`max_row`	Maximum number of rows to have on a page. Spillover will hang over the side of the page..
`max_col`	Maximum number of columns to have on a page. Spillover will be wrapped to multiple pages.

`notes_detect`	Vector of strings to detect in output tables (example `c("NC", "BLQ")`).
`rows_by`	Can be either "time" or "id". If it is "time", there will be a column for time and separate column for each subject ID. If rows_by is set to "id" there will be a column for ID and a column for each individual time.

Value

List containing the following elements

- isgood: Boolean indicating the exit status of the function.
- one_table: Dataframe of the entire table with the first lines containing the header.
- one_body: Dataframe of the entire table (data only).
- one_header: Dataframe of the entire header (row and body, no data).
- tables: Named list of tables. Each list element is of the output
- msgs: Vector of text messages describing any errors that were found. format from `build_span`.

Examples

```
# We need a state variable to be define
sess_res = NCA_test_mksession()

state = sess_res$state

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This is the raw PKNCA output
pknca_res = NCA_fetch_ana_pknca(state, current_ana)

# Building the figure
mk_res = mk_table_ind_obs(nca_res = pknca_res)
mk_res$tables[["Table 1"]]$ft
```

`mk_table_nca_params` *Create Tabular Output from PKNCA Results*

Description

Create paginated tables from PKNCA to use in reports and Shiny apps.

Usage

```
mk_table_nca_params(
  nca_res,
  type = "individual",
  grouping = "interval",
  not_calc = "NC",
```

```
  obnd = NULL,
  nps = NULL,
  mult_str = "*",
  infinity = "inf",
  digits = NULL,
  text_format = "text",
  notes_detect = NULL,
  max_height = 7,
  max_width = 6.5,
  max_row = NULL,
  max_col = NULL
)
```

Arguments

`nca_res`	Output of PKNCA.
`type`	Type of table to generate. Can be either `"individual"` or `"summary"]`.
`grouping`	How to group columns in tables. Can be either `"interval"` or `"parameter"]`.
`not_calc`	Text string to replace NA values with to indicated values were not calculated.
`obnd`	onbrand reporting object.
`nps`	NCA parameter summary table with the following columns. - parameter: PKNCA Paramter name. - text: Name used in text output. - md: Name used markdown output. - latex: Name used in latex output. - description: Verbose textual description of the parameter.
`mult_str`	Text string to replace * values in units.
`infinity`	Text string to replace infinity in time intervals in column headers.
`digits`	Number of significant figures to report (set to `NULL` to disable rounding)
`text_format`	Either `"md"` for markdown or `"text"` (default) for plain text.
`notes_detect`	Vector of strings to detect in output tables (example `c("NC", "BLQ")`).
`max_height`	Maximum height of the final table in inches (A value of `NULL` will use 100 inches).
`max_width`	Maximum width of the final table in inches (A value of `NULL` will use 100 inches).
`max_row`	Maximum number of rows to have on a page. Spillover will hang over the side of the page..
`max_col`	Maximum number of columns to have on a page. Spillover will be wrapped to multiple pages.

Value

list containing the following elements

- raw_nca: Raw PKNCA output.
- isgood: Boolean indicating the exit status of the function.
- one_table: Dataframe of the entire table with the first lines containing the header.
- one_body: Dataframe of the entire table (data only).
- one_header: Dataframe of the entire header (row and body, no data).
- tables: Named list of tables. Each list element is of the output
- msgs: Vector of text messages describing any errors that were found. format from `build_span`.

Examples

```
# We need a state variable to be define
sess_res = NCA_test_mksession()

state = sess_res$state

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This is the raw PKNCA output
pknca_res = NCA_fetch_ana_pknca(state, current_ana)

# Parameter reporting details from the ruminate configuration
nps  = state[["NCA"]][["nca_parameters"]][["summary"]]

# Building the figure
mk_res = mk_table_nca_params(nca_res = pknca_res, nps=nps, digits=3)
mk_res$tables[["Table 1"]]$ft
```

`NCA_add_int` *Adds Analysis Interval to Current Analysis*

Description

Takes the start time, stop time, and NCA parameters and adds them to the intervals table of the current analysis

Usage

```
NCA_add_int(state, interval_start, interval_stop, nca_parameters)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`
`interval_start`	Interval start time (numeric).
`interval_stop`	Interval stop time (numeric).
`nca_parameters`	list of NCA parameters in the interval

Value

State with interval added to the current analysis.

`NCA_append_report` *Append Report Elements*

Description

Takes an NCA state object and appends any reportable elements for the specified report type. On NCA analyses that are in a "good" state will be reported. Those not in a good state will be ignored.

Usage

```
NCA_append_report(state, rpt, rpttype, gen_code_only = FALSE)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`
`rpt`	Report with the current content of the report which will be appended to in this function. For details on the structure see the documentation for `formods::FM_generate_report()`
`rpttype`	Type of report to generate (supported "xlsx", "pptx", "docx").
`gen_code_only`	Boolean value indicating that only code should be generated (`FALSE`).

Value

list containing the following elements

- isgood: Return status of the function.
- hasrptele: Boolean indicator if the module has any reportable elements.
- code: Code to create report elements.
- msgs: Messages to be passed back to the user.
- rpt: Report with any additions passed back to the user.

See Also

`formods::FM_generate_report()`

Examples

```
# We need a state object to use below
sess_res = NCA_test_mksession()
state = sess_res$state

# here we need an empty report object for tabular data
rpt = list(summary = list(), sheets=list())

# Now we append the report indicating we want
# Excel output:
rpt_res = NCA_append_report(state,
  rpt           = rpt,
  rpttype       = "xlsx",
  gen_code_only = TRUE)

# Shows if report elements are present
rpt_res$hasrptele

# Code chunk to generate report element
cat(paste(rpt_res$code, collapse="\n"))
```

`nca_builder` *Builds NCA Code from ui Elements*

Description

Takes the current analysis in the state object and creates the code to run the analysis

Usage

```
nca_builder(state)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`

Value

NCA state with the NCA for the current analysis built.

Examples

```
# We need a module variables to be defined
sess_res = NCA_test_mksession()

state = sess_res$state

state = nca_builder(state)
```

`NCA_fetch_ana_ds` *Fetch Analysis Dataset*

Description

Fetches the dataset used for the specified analysis

Usage

```
NCA_fetch_ana_ds(state, current_ana)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`
`current_ana`	Current value in the analysis

Value

Dataset from the ds field of FM_fetch_ds()

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id             = id,
                        input          = input,
                        session        = session,
                        FM_yaml_file   = FM_yaml_file,
                        MOD_yaml_file  = MOD_yaml_file,
                        id_ASM         = id_ASM,
                        id_UD          = id_UD,
                        id_DW          = id_DW,
```

```
                        react_state     = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)

# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
id_col = NCA_find_col(
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
state = NCA_new_ana(state)
```

`NCA_fetch_ana_pknca` *Fetch PKNCA Results Object*

Description

Fetches the PKNCA output for a specified analysis

Usage

```
NCA_fetch_ana_pknca(state, current_ana)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`
`current_ana`	Current value in the analysis

Value

Dataset from the `ds` field of FM_fetch_ds()

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id             = id,
                       input          = input,
                       session        = session,
                       FM_yaml_file   = FM_yaml_file,
                       MOD_yaml_file  = MOD_yaml_file,
                       id_ASM         = id_ASM,
                       id_UD          = id_UD,
                       id_DW          = id_DW,
                       react_state    = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)

# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
id_col = NCA_find_col(
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
```

```
state = NCA_new_ana(state)
```

`NCA_fetch_code` *Fetch Module Code*

Description

Fetches the code to generate results seen in the app

Usage

```
NCA_fetch_code(state)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`

Value

Character object vector with the lines of code

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id             = id,
                        input          = input,
                        session        = session,
                        FM_yaml_file   = FM_yaml_file,
                        MOD_yaml_file  = MOD_yaml_file,
                        id_ASM         = id_ASM,
                        id_UD          = id_UD,
                        id_DW          = id_DW,
```

```
                        react_state     = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)

# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
id_col = NCA_find_col(
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
state = NCA_new_ana(state)
```

`NCA_fetch_current_ana` *Fetches Current Analysis*

Description

Takes an NCA state and returns the current active analysis

Usage

```
NCA_fetch_current_ana(state)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`

Value

List containing the details of the current analysis. The structure of this list is the same as the structure of `state$NCA$anas` in the output of `NCA_fetch_state()`.

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id             = id,
                       input           = input,
                       session         = session,
                       FM_yaml_file    = FM_yaml_file,
                       MOD_yaml_file   = MOD_yaml_file,
                       id_ASM          = id_ASM,
                       id_UD           = id_UD,
                       id_DW           = id_DW,
                       react_state     = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)

# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
id_col = NCA_find_col(
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
```

```
state = NCA_new_ana(state)
```

`NCA_fetch_current_obj` *Fetches the Current Analysis Object*

Description

Takes the current state and object type and returns the currently selected object. For example if you have specified figure, it will look at the output figure selected and the figure number of that figure and return the ggplot object for that. by subject id highlighting of certain NCA aspects (e.g. points used for half-life)

Usage

```
NCA_fetch_current_obj(state, obj_type)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`
`obj_type`	Type of object to return (either "table" or "figure").

Value

List with a format that depends on the obj_type. For figures:

- ggplot: ggplot object of the figure.
- isgood: Return status of the function.
- msgs: Messages to be passed back to the user.

For tables:

- df: Dataframe of the current table.
- ft: Flextable object of the current table.
- notes: Any table notes to be included.
- isgood: Return status of the function.
- msgs: Messages to be passed back to the user.

Examples

```
# We need a state object to use below
sess_res = NCA_test_mksession()
state = sess_res$state

# Current active table:
res = NCA_fetch_current_obj(state, "table")
res$ft

# Current active figure:
res = NCA_fetch_current_obj(state, "figure")
res$ggplot
```

`NCA_fetch_data_format` *Fetches Details About Data Requirements*

Description

Use this to get information about data formats.

Usage

```
NCA_fetch_data_format(
  MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")
)
```

Arguments

`MOD_yaml_file`	Module configuration file with MC as main section.

Value

List with details about the data formats

Examples

```
NCA_fetch_data_format()
```

`NCA_fetch_ds` *Fetch Module Datasets*

Description

Fetches the datasets contained in the module

Usage

```
NCA_fetch_ds(state)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`

Value

list containing the following elements

- isgood: Return status of the function.
- hasds: Boolean indicator if the module has any datasets
- msgs: Messages to be passed back to the user.
- ds: List with datasets. Each list element has the name of the R-object for that dataset. Each element has the following structure:
  - label: Text label for the dataset
  - MOD_TYPE: Short name for the type of module.
  - id: module ID
  - DS: Dataframe containing the actual dataset.
  - DSMETA: Metadata describing DS
  - code: Complete code to build dataset.
  - checksum: Module checksum.
  - DSchecksum: Dataset checksum.

Examples

```
# We need a state object to use below
sess_res = NCA_test_mksession()
state = sess_res$state

myDs = NCA_fetch_ds(state)
```

`NCA_fetch_np_meta` *Fetches NCA Parameter Meta Information*

Description

This provides meta information about NCA parameters. This includes parameter names, text descriptions, formatting (md and LaTeX).

Usage

```
NCA_fetch_np_meta(
  MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")
)
```

Arguments

`MOD_yaml_file`	Module configuration file with MC as main section.

Value

List with the following elements:

- choices: List parameter choices grouped by values specified in the module configuration file.
- summary: Data frame with meta data about the NCA parameters with the following columns:
  - parameter: Name of parameter in PKNCA.
  - text: Name of parameter in plain text.
  - md: Parameter name formatted in Markdown.
  - latex: Parameter name formatted using LaTeX.
  - description: Verbose description in plain text for the parameter.

Examples

```
NCA_fetch_np_meta()
```

`NCA_fetch_PKNCA_meta` *Fetches PKNCA Metadata*

Description

Compiles Metadata from PKNCA

Usage

```
NCA_fetch_PKNCA_meta()
```

Value

Dataframe containing PKCNA metadata for NCA parameters.

Examples

```
PKNCA_meta = NCA_fetch_PKNCA_meta()
utils::head(PKNCA_meta)
```

`NCA_fetch_state` *Fetch ruminate State*

Description

Merges default app options with the changes made in the UI

Usage

```
NCA_fetch_state(
  id,
  input,
  session,
  FM_yaml_file,
  MOD_yaml_file,
  id_ASM,
  id_UD,
  id_DW,
  react_state
)
```

Arguments

`id`	Shiny module ID
`input`	Shiny input variable
`session`	Shiny session variable
`FM_yaml_file`	App configuration file with FM as main section.
`MOD_yaml_file`	Module configuration file with MC as main section.
`id_ASM`	ID string for the app state management module used to save and load app states
`id_UD`	ID string for the upload data module used to save and load app states
`id_DW`	ID string for the data wrangling module used to save and load app states
`react_state`	Variable passed to server to allow reaction outside of module (`NULL`)

Value

list containing the current state of the app including default values from the yaml file as well as any changes made by the user. The list has the following structure:

- yaml: Full contents of the supplied yaml file.
- MC: Module components of the yaml file.
- NCA:
  - ana_cntr: Analysis counter.
  - anas: List of analyses: Each analysis has the following structure:

    - ana_dsview: Dataset view/ID (name from DSV) selected as a data source for this analysis.
    - ana_scenario: Analysis scenario selected in the UI
    - checksum: checksum of the analysis (used to detect changes in the analysis).
    - code: Code to generate analysis from start to finish or error messages if code generation/analysis failed.
    - code_components: List containing the different components from code
    - col_conc: Column from ana_dsview containing the concentration data.
    - col_dose: Column from ana_dsview containing the dose amount.
    - col_dur: Column from ana_dsview containing the infusion duration or N/A if unused.
    - col_group: Columns from ana_dsview containing other grouping variables.
    - col_id: Column from ana_dsview containing the subject IDs.
    - col_ntime: Column from ana_dsview containing the nominal time values
    - col_route: Column from ana_dsview containing the dosing route.
    - col_time: Column from ana_dsview containing the time values.
    - id: Character id (`ana_idx`).
    - idx: Numeric id (`1`).
    - include_units: Boolean variable indicating in units should included in the analysis.
    - interval_range: Vector with the first element representing he beginning of the interval and the second element containing the end of the interval.
    - intervals: List of the intervals to include.
    - isgood: Current status of the analysis.
    - key: Analysis key acts as a title/caption (user editable)
    - msgs: Messages generated when checking configuration and analysis options.
    - nca_config: List of NCA configuration options for this analysis.
    - nca_object_name: Prefix for NCA objects associated with this analyis.
    - nca_parameters: NCA parameters selected for calculation in the UI.
    - notes: Analysis notes (user editable)
    - objs: List of names and values for objects created with generated code.
    - sampling: Sampling method either "sparse" or "serial"
    - units_amt: Amount units.
    - units_conc: Concentration units.
    - units_dose: Dosing units.
    - units_time: Time units.
  - current_ana: Currently selected analysis (list name element from anas).
  - DSV: Available data source views (see `FM_fetch_ds`)
  - checksum: This is an MD5 sum of the module (checksum of the analysis checksums).
  - nca_config: List of PKNCA configuration options for this analysis.
  - nca_parameters: List with two elements
    - choices: List consisting of "Common Parameters" and "Other" (used for grouping in the UI). Each of these is a list of text parameter names with a value of the PKNCA parameter name.
    - summary: Summary table with the following columns:

        - parameter: PKNCA Paramter name.
        - text: Name used in text output.
        - md: Name used markdown output.
        - latex: Name used in latex output.
        - description: Verbose textual description of the parameter.
    - ui: Current value of form elements in the UI.
    - ui_ana_map: Map between UI element names and analysis in the object you get from `NCA_fetch_current_ana`
    - ui_ids: Vector of UI elements for the module.
    - ui_hold: List of hold elements to disable updates before a full ui referesh is complete.
- MOD_TYPE: Character data containing the type of module `"NCA"`
- id: Character data containing the module id module in the session variable.
- FM_yaml_file: App configuration file with FM as main section.
- MOD_yaml_file: Module configuration file with MC as main section.

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id            = id,
                       input          = input,
                       session        = session,
                       FM_yaml_file   = FM_yaml_file,
                       MOD_yaml_file  = MOD_yaml_file,
                       id_ASM         = id_ASM,
                       id_UD          = id_UD,
                       id_DW          = id_DW,
                       react_state    = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)
```

```
# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)

# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
id_col = NCA_find_col(
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
state = NCA_new_ana(state)
```

`NCA_find_col` *Determines Default Column Name*

Description

Based on the current analysis, value from the UI, an optional list of patterns to search, and column names from a dataset this function tries to find a default value for a column in the analysis (e.g. subject id, dose, concentration, etc).

Generally the following is done:

- If curr_ui has a non-NULL, non-"" value it is compared to dscols. If it is found there that value is returned.
- If not then the patterns are considered. If the patterns from the YAML file are not NULL they are compared sequentially to the columns names. The first match found is returned.
- If nothing is found then the first value of dscols is returned.

Usage

```
NCA_find_col(
  curr_ana = NULL,
  curr_ui = NULL,
  patterns = NULL,
  dscols,
  null_ok = FALSE
)
```

Arguments

`curr_ana`	Current value in the analysis
`curr_ui`	Current value in UI
`patterns`	List of regular expression patterns to consider.
`dscols`	Columns from the dataset.
`null_ok`	Logical value indicating if a null result (nothing found) is OK (default: `FALSE`)

Value

Name of column found based on the rules above.

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id             = id,
                       input           = input,
                       session         = session,
                       FM_yaml_file    = FM_yaml_file,
                       MOD_yaml_file   = MOD_yaml_file,
                       id_ASM          = id_ASM,
                       id_UD           = id_UD,
                       id_DW           = id_DW,
                       react_state     = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)
```

```
# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
id_col = NCA_find_col(
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
state = NCA_new_ana(state)
```

`NCA_init_state` *Initialize NCA Module State*

Description

Creates a list of the initialized module state

Usage

```
NCA_init_state(FM_yaml_file, MOD_yaml_file, id, id_UD, id_DW, session)
```

Arguments

`FM_yaml_file`	App configuration file with FM as main section.
`MOD_yaml_file`	Module configuration file with MC as main section.
`id`	ID string for the module.
`id_UD`	ID string for the upload data module used to handle uploads or the name of the list element in react_state where the data set is stored.
`id_DW`	ID string for the data wrangling module to process any uploaded data
`session`	Shiny session variable (in app) or a list (outside of app)

Value

list containing an empty NCA state

`NCA_load_scenario` *Loads Pre-Defined Scenario*

Description

Loads a pre-defined analysis scneario from the NCA YAML config file.

Usage

```
NCA_load_scenario(state, ana_scenario)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`
`ana_scenario`	Short name of the analysis scenario to load from the config file.

Value

NCA state object with the scenario loaded and relevant notifications set.

`NCA_mkactive_ana` *Fetch PKNCA Results Object*

Description

Fetches the PKNCA output for a specified analysis

Usage

```
NCA_mkactive_ana(state, ana_id)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`
`ana_id`	Analysis ID to make active.

Value

State with the analysis ID made active. JMH add to example script below

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id             = id,
                       input           = input,
                       session         = session,
                       FM_yaml_file    = FM_yaml_file,
                       MOD_yaml_file   = MOD_yaml_file,
                       id_ASM          = id_ASM,
                       id_UD           = id_UD,
                       id_DW           = id_DW,
                       react_state     = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)

# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
id_col = NCA_find_col(
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
```

```
state = NCA_new_ana(state)
```

`NCA_mk_preload` *Make List of Current NCA State*

Description

Reads in the app state from yaml files.

Usage

```
NCA_mk_preload(state)
```

Arguments

`state`	NCA state object

Value

list with the following elements

- isgood: Boolean indicating the exit status of the function.
- msgs: Messages to be passed back to the user.
- yaml_list: Lists with preload components.

Examples

```
sess_res = NCA_test_mksession()
state = sess_res$state
res = NCA_mk_preload(state)
```

`NCA_new_ana` *Initialize New Analysis*

Description

Creates a new NCA analysis in an NCA module

Usage

```
NCA_new_ana(state)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`

Value

NCA state object containing a new empty analysis and that analysis is set as the current active analyisis

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id             = id,
                       input          = input,
                       session        = session,
                       FM_yaml_file   = FM_yaml_file,
                       MOD_yaml_file  = MOD_yaml_file,
                       id_ASM         = id_ASM,
                       id_UD          = id_UD,
                       id_DW          = id_DW,
                       react_state    = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)

# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
```

```
id_col = NCA_find_col(
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
state = NCA_new_ana(state)
```

`NCA_preload` *Preload Data for NCA Module*

Description

Populates the supplied session variable with information from list of sources.

Usage

```
NCA_preload(
  session,
  src_list,
  yaml_res,
  mod_ID = NULL,
  react_state = list(),
  quickload = FALSE
)
```

Arguments

`session`	Shiny session variable (in app) or a list (outside of app)
`src_list`	List of preload data (all read together with module IDs at the top level)
`yaml_res`	List data from module yaml config
`mod_ID`	Module ID of the module being loaded.
`react_state`	Reactive shiny object (in app) or a list (outside of app) used to trigger reactions.
`quickload`	Logical `TRUE` to load reduced analysis `FALSE` to load the full analysis

Value

list with the following elements

- isgood: Boolean indicating the exit status of the function.
- msgs: Messages to be passed back to the user.
- session: Session object
- input: The value of the shiny input at the end of the session initialization.
- state: App state.
- react_state: The `react_state` components.

`NCA_process_current_ana` *Processes Current Analysis to be Run*

Description

Takes the current analysis and checks different aspects to for any issues to make sure it's good to go.

Usage

```
NCA_process_current_ana(state)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`

Value

Current analysis list with isgood and msgs set

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id             = id,
                        input          = input,
                        session        = session,
                        FM_yaml_file    = FM_yaml_file,
                        MOD_yaml_file   = MOD_yaml_file,
                        id_ASM         = id_ASM,
                        id_UD          = id_UD,
                        id_DW          = id_DW,
```

```
                         react_state     = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)

# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
id_col = NCA_find_col(
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
state = NCA_new_ana(state)
```

`NCA_Server` *Fetch Non-Compartmental Analysis State*

Description

Merges default app options with the changes made in the UI

Usage

```
NCA_Server(
  id,
  FM_yaml_file = system.file(package = "formods", "templates", "formods.yaml"),
  MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml"),
  id_ASM = "ASM",
  id_UD = "UD",
  id_DW = "DW",
  deployed = FALSE,
  react_state = NULL
)
```

Arguments

`id`	An ID string that corresponds with the ID used to call the modules UI elements

`FM_yaml_file`	App configuration file with FM as main section.
`MOD_yaml_file`	Module configuration file with MC as main section.
`id_ASM`	ID string for the app state management module used to save and load app states
`id_UD`	ID string for the upload data module used to save and load app states
`id_DW`	ID string for the data wrangling module used to save and load app states
`deployed`	Boolean variable indicating whether the app is deployed or not.
`react_state`	Variable passed to server to allow reaction outside of module (`NULL`)

Value

list containing the current state of the app including default values from the yaml file as well as any changes made by the user. The list has the following structure:

- yaml: Full contents of the supplied yaml file.
- MC: Module components of the yaml file.
- NCA:
  - isgood: Boolean object indicating if the file was successfully loaded.
  - checksum: This is an MD5 sum of the contents element and can be used to detect changes in the state.
- MOD_TYPE: Character data containing the type of module `"NCA"`
- id: Character data containing the module id module in the session variable.
- FM_yaml_file: App configuration file with FM as main section.
- MOD_yaml_file: Module configuration file with MC as main section.

Examples

```
if(interactive()){
# original file: inst/templates/ruminate.R
library(formods)
library(ruminate)

# These are suggested packages
library(shinydashboard)
#library(ggpubr)
#library(plotly)
#library(shinybusy)
library(prompter)
#library(utils)

tags$style("@import url(https://use.fontawesome.com/releases/v6.4.0/css/all.css);")

# You can copy these locally and customize them for your own needs. Simply
# change the assignment to the local copy you've modified.
formods.yaml  = system.file(package="formods",  "templates",  "formods.yaml")
ASM.yaml      = system.file(package="formods",  "templates",  "ASM.yaml")
UD.yaml       = system.file(package="formods",  "templates",  "UD.yaml")
DW.yaml       = system.file(package="formods",  "templates",  "DW.yaml")
```

```
FG.yaml       = system.file(package="formods",  "templates",  "FG.yaml")
NCA.yaml      = system.file(package="ruminate", "templates",  "NCA.yaml")


# Name of  file to indicate we need to load testing data
ftmptest = file.path(tempdir(), "ruminate.test")


# Making sure that the deployed object is created
if(!exists("deployed")){
  deployed = FALSE
}

# If the DEPLOYED file marker existrs we set deployed to TRUE
if(file.exists("DEPLOYED")){
  deployed = TRUE
}

# # Making sure that the run_dev object is created
# if(file.exists(file.path(tempdir(), "RUMINTE_DEVELOPMENT"))){
#   run_dev  = TRUE
# }else{
#   run_dev  = FALSE
# }

# If the SETUP.R file exists we source it
if(file.exists("SETUP.R")){
  source("SETUP.R")
}



CSS <- "
.wrapfig {
  float: right;
  shape-margin: 20px;
  margin-right: 20px;
  margin-bottom: 20px;
}
"

#https://fontawesome.com/icons?from=io
logo_url =
  "https://raw.githubusercontent.com/john-harrold/ruminate/main/man/figures/logo.png"
data_url =
  "https://github.com/john-harrold/formods/raw/master/inst/test_data/TEST_DATA.xlsx"
run_url  =
  "https://runruminate.ubiquity.tools/"
use_url  =
  "https://useruminate.ubiquity.tools/"
main_url =
  "https://ruminate.ubiquity.tools/"
issue_url =
```

```
  "https://github.com/john-harrold/ruminate/issues"

intro_text = tags$p(
"Ruminate is a shiny module for pharmacometric data processing,
visualization, and analysis. It consists of separate shiny modules that
provide interfaces into common R packages and provides the underlying code.
This is done to facilitate usage of those packages and to provide reproducible
analyses." ,
tags$li( "To find out more visit  ",
        tags$a("ruminate.ubiquity.tools", href=main_url),""),
tags$li( "To give it a try you can download a test dataset ",
        tags$a("here", href=data_url),""),
tags$li( "Go to  ",
        tags$a("useruminate.ubiquity.tools", href=use_url)," for a video
        demonstrating how to use ruminate"),
tags$li( "If you run into any problems, have questions, or want a feature please
        visit the ",
        tags$a("issues", href=issue_url)," page")
)


ui <- shinydashboard::dashboardPage(
  skin="black",
  shinydashboard::dashboardHeader(title="ruminate"),
  shinydashboard::dashboardSidebar(
     shinydashboard::sidebarMenu(
       shinydashboard::menuItem("Load/Save",
                                tabName="loadsave",
                                icon=icon("arrow-down-up-across-line")) ,
     shinydashboard::menuItem("Transform Data", tabName="wrangle", icon=icon("shuffle")),
     shinydashboard::menuItem("Visualize",    tabName="plot",   icon=icon("chart-line")),
      shinydashboard::menuItem("NCA",         tabName="nca",    icon=icon("chart-area")),
     shinydashboard::menuItem("App Info",     tabName="sysinfo", icon=icon("book-medical"))
     )
  ),
  shinydashboard::dashboardBody(
  tags$head(
    tags$style(HTML(CSS))
  ),
    shinydashboard::tabItems(
       shinydashboard::tabItem(tabName="nca",
               shinydashboard::box(title="Non-Compartmental Analysis", width=12,
               fluidRow( prompter::use_prompt(),
               column(width=12,
               htmlOutput(NS("NCA",  "NCA_ui_compact")))))
               ),
       shinydashboard::tabItem(tabName="loadsave",
         #     shinydashboard::box(title=NULL, width=12,
               shinydashboard::tabBox(
                 width = 12,
                 title = NULL,
                 shiny::tabPanel(id="load_data",
                          title=tagList(shiny::icon("file-arrow-up"),
```

```
                                "Load Data"),
                fluidRow(
                  column(width=6,
                    div(style="display:inline-block;width:100%",
                    htmlOutput(NS("UD", "ui_ud_load_data"))),
                    htmlOutput(NS("UD", "ui_ud_clean")),
                    htmlOutput(NS("UD", "ui_ud_select_sheets")),
                    htmlOutput(NS("UD", "ui_ud_workflows")),
                    htmlOutput(NS("UD", "ui_ud_text_load_result"))),
                  column(width=6,
                      tags$p(
                          tags$img(
                          class = "wrapfig",
                          src   = logo_url,
                          width = 150,
                          alt = "formods logo" ),
                      intro_text
                      ))
                ),
              fluidRow(
                column(width=12,
                       div(style="display:inline-block;vertical-align:top",
                                 htmlOutput(NS("UD", "ui_ud_data_preview")))
                       ))
              ),
              shiny::tabPanel(id="save_state",
                       title=tagList(shiny::icon("arrow-down-up-across-line"),
                                     "Save or Load Analysis"),
              fluidRow(
                column(width=5,
                       div(style="display:inline-block;vertical-align:top",
                htmlOutput(NS("ASM", "ui_asm_compact"))
                ))
                )
              )
            )
  #         ),
            ),
shinydashboard::tabItem(tabName="wrangle",
    shinydashboard::box(title="Transform and Create Views of Your Data", width=12,
        fluidRow(
        column(width=12,
        htmlOutput(NS("DW",  "DW_ui_compact")))))
        ),
shinydashboard::tabItem(tabName="plot",
        shinydashboard::box(title="Visualize Data", width=12,
        htmlOutput(NS("FG",  "FG_ui_compact")))),
shinydashboard::tabItem(tabName="sysinfo",
  #     box(title="System Details", width=12,
        shinydashboard::tabBox(
          width = 12,
          title = NULL,
          shiny::tabPanel(id="sys_packages",
```

```
                        title=tagList(shiny::icon("box-open"),
                                      "Installed Packages"),
               htmlOutput(NS("ASM", "ui_asm_sys_packages"))
               ),
               shiny::tabPanel(id="sys_modules",
                        title=tagList(shiny::icon("cubes"),
                                      "Loaded Modules"),
               htmlOutput(NS("ASM", "ui_asm_sys_modules"))
               ),
               shiny::tabPanel(id="sys_log",
                        title=tagList(shiny::icon("clipboard-list"),
                                      "Log"),
               verbatimTextOutput(NS("ASM", "ui_asm_sys_log"))
               ),
               shiny::tabPanel(id="sys_options",
                        title=tagList(shiny::icon("sliders"),
                                      "R Options"),
               htmlOutput(NS("ASM", "ui_asm_sys_options"))
               )
       #       )
              ))
     )
    )
   )

 # Main app server
 server <- function(input, output, session) {

   # Empty reactive object to track and react to
   # changes in the module state outside of the module
   react_FM = reactiveValues()

   # Module IDs and the order they are needed for code generation
   mod_ids = c("UD", "DW", "FG", "NCA", "MB")

   # If the ftmptest file is present we load test data
   if(file.exists(ftmptest)){
     sources = c(system.file(package="formods",  "preload", "ASM_preload.yaml"),
                 system.file(package="formods",  "preload", "UD_preload.yaml"),
                 system.file(package="formods",  "preload", "FG_preload.yaml"),
                 system.file(package="formods",  "preload", "DW_preload.yaml"),
                 system.file(package="ruminate", "preload", "NCA_preload.yaml"))

     res = FM_app_preload(session=session, sources=sources)
   # Otherwise we look for a preload file and load that if it exists
   } else if(file.exists("preload.yaml")){
    shinybusy::show_modal_spinner(text="Preloading analysis, be patient", session=session)
     res = FM_app_preload(session=session, sources="preload.yaml")
     shinybusy::remove_modal_spinner(session = session)
   }

   # Module servers
   formods::ASM_Server( id="ASM",
```

```
                       deployed      = deployed,
                       react_state   = react_FM,
                       FM_yaml_file  = formods.yaml,
                       MOD_yaml_file = ASM.yaml,
                       mod_ids       = mod_ids)
  formods::UD_Server(  id ="UD", id_ASM = "ASM",
                       deployed         = deployed,
                       react_state      = react_FM,
                       MOD_yaml_file    = UD.yaml,
                       FM_yaml_file     = formods.yaml)
  formods::DW_Server(  id="DW",       id_ASM = "ASM",
                       id_UD = "UD",
                       deployed         = deployed,
                       react_state      = react_FM,
                       MOD_yaml_file    = DW.yaml,
                       FM_yaml_file     = formods.yaml)
  formods::FG_Server(  id="FG",     id_ASM = "ASM",
                       id_UD = "UD", id_DW = "DW",
                       deployed         = deployed,
                       react_state      = react_FM,
                       MOD_yaml_file    = FG.yaml,
                       FM_yaml_file     = formods.yaml)
  ruminate::NCA_Server(id    ="NCA", id_ASM = "ASM",
                       id_UD = "UD", id_DW  = "DW",
                       deployed         = deployed,
                       react_state      = react_FM,
                       MOD_yaml_file    = NCA.yaml,
                       FM_yaml_file     = formods.yaml)

}

shinyApp(ui, server)
}
```

`NCA_set_current_ana` *Sets Current Analysis*

Description

Takes an NCA state and an analysis list and sets that figure list as the value for the active figure

Usage

```
NCA_set_current_ana(state, ana)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`
`ana`	Analysis list from `NCA_fetch_current_ana`

Value

State with the current analysis updated

Examples

```
library(ruminate)
# Module IDs
id     = "NCA"
id_UD  = "UD"
id_DW  = "DW"
id_ASM = "ASM"
# We need session and input variables to be define
 sess_res = NCA_test_mksession()

# Extracting the session and input variables
session      = sess_res$session
input        = sess_res$input
react_state  = list()

# We also need configuration files
FM_yaml_file  = system.file(package = "formods",  "templates", "formods.yaml")
MOD_yaml_file = system.file(package = "ruminate", "templates", "NCA.yaml")

# Getting the current module state
state = NCA_fetch_state(id             = id,
                       input          = input,
                       session        = session,
                       FM_yaml_file   = FM_yaml_file,
                       MOD_yaml_file  = MOD_yaml_file,
                       id_ASM         = id_ASM,
                       id_UD          = id_UD,
                       id_DW          = id_DW,
                       react_state    = react_state)

# Pulls out the active analysis
current_ana = NCA_fetch_current_ana(state)

# This will get the dataset associated with this analysis
ds = NCA_fetch_ana_ds(state, current_ana)

# After making changes you can update those in the state
state = NCA_set_current_ana(state, current_ana)

# You can use this to check the current analysis
current_ana = NCA_process_current_ana(state)

# This will pull out the code for the module
fc_res = NCA_fetch_code(state)

# This will use patterns defined for the site to detect
# columns. In this example we are detecting the id column:
id_col = NCA_find_col(
```

```
  patterns = state[["MC"]][["detect_col"]][["id"]],
  dscols   = names(ds$DS))

# This creates a new analysis
state = NCA_new_ana(state)
```

`NCA_test_mksession` *Populate Session Data for Module Testing*

Description

Populates the supplied session variable for testing.

Usage

```
NCA_test_mksession(session = list(), full = FALSE)
```

Arguments

`session`	Shiny session variable (in app) or a list (outside of app)
`full`	Boolean indicating if the full test session should be created

Value

The NCA portion of the `all_sess_res` returned from `FM_app_preload`

See Also

`FM_app_preload`

Examples

```
session = shiny::MockShinySession$new()
sess_res = NCA_test_mksession(session=session)
```

`plot_sr_ev` *Plots Timecourse of Rules Simulations*

Description

```
Plots the timecourse of `simulate_rules()` output.
```

Usage

```
plot_sr_ev(
  sro = NULL,
  fpage = 1,
  fcol = "id",
  xcol = "time",
  error_msgs = NULL,
  ylog = TRUE,
  ylab_str = "Amount",
  xlab_str = "Time",
  post_proc = "fig  = fig + ggplot2::theme_light()",
  evplot = c(1, 4),
  fncol = 4,
  fnrow = 2
)
```

Arguments

`sro`	Output of 'simulate_rules()'.
`fpage`	If facets are selected and multiple pages are generated then this indcates the page to return.
`fcol`	Name of column to facet by or `NULL` to disable faceting (`"id"`).
`xcol`	Name of column to take x-data from (`"time"`).
`error_msgs`	Named list with error messages to overwrite (`NULL`
`ylog`	Boolean to enable log10 scaling of the y-axis (`TRUE`
`ylab_str`	Label for the y-axis (`"Output"`
`xlab_str`	Label for the x-axis (`"Output"`
`post_proc`	Character object with post processing post-processing code for the figure object named `fig` internall (`"fig = fig + theme_light()"`)
`evplot`	Evids to plot can be 1 or 4
`fncol`	Number of columns in faceted output.
`fnrow`	Number of rows in faceted output.

Details

For a detailed examples see `vignette("clinical_trial_simulation", package = "ruminate")`.

Value

List with the followin1g elements:

- isgood: Return status of the function.
- msgs: Error or warning messages if any issues were encountered.
- npages: Total number of pages using the current configuration.
- error_msgs: List of error messages used.
- dsp: Intermediate dataset generated from `sro` to plot in ggplot.
- fig: Figure generated.

Examples

```
library(formods)
library(ggplot2)

# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html

# None of this will work if rxode2 isn't installed:
if(is_installed("rxode2")){
library(rxode2)
set.seed(8675309)
rxSetSeed(8675309)

my_model = function ()
{
   description <- "One compartment PK model with linear clearance using differential equations"
    ini({
        lka <- 0.45
        label("Absorption rate (Ka)")
        lcl <- 1
        label("Clearance (CL)")
        lvc <- 3.45
        label("Central volume of distribution (V)")
        propSd <- c(0, 0.5)
        label("Proportional residual error (fraction)")
        etalcl ~ 0.1
    })
    model({
        ka <- exp(lka)
        cl <- exp(lcl + etalcl)
        vc <- exp(lvc)
        kel <- cl/vc
        d/dt(depot) <- -ka * depot
        d/dt(central) <- ka * depot - kel * central
        Cc <- central/vc
        Cc ~ prop(propSd)
    })
}
```

```
# This creates an rxode2 object
object  = rxode(my_model)

# If you want details about the parameters, states, etc
# in the model you can use this:
rxdetails = fetch_rxinfo(object)

rxdetails$elements

# Next we will create subjects. To do that we need to
# specify information about covariates:
nsub = 2
covs = list(
  WT         = list(type     = "continuous",
                    sampling = "log-normal",
                    values   = c(70, .15))
)

subs = mk_subjects(object = object,
                   nsub   = nsub,
                   covs   = covs)

head(subs$subjects)

rules = list(
  dose = list(
    condition = "TRUE",
    action    = list(
      type  = "dose",
      state     = "central",
      values    = "c(1)",
      times     = "c(0)",
      durations = "c(0)")
    )
)

# We evaulate the rules for dosing at time 0
eval_times =  0

# Stop 2 months after the last dose
output_times = seq(0, 56, 1)

# This runs the rule-based simulations
simres =
  simulate_rules(
    object        = object,
    subjects      = subs[["subjects"]],
    eval_times    = eval_times,
    output_times  = output_times,
    rules         = rules)

# First subject data:
sub_1 = simres$simall[simres$simall$id == 1, ]
```

```
# First subjects events
evall = as.data.frame(simres$evall)
ev_sub_1 = evall[evall$id ==1, ]

# All of the simulation data
simall = simres$simall
simall$id = as.factor(simall$id)

# Timecourse
psim =
  plot_sr_tc(
    sro    = simres,
    dvcols = "Cc")
psim$fig

# Events
pev =
  plot_sr_ev(
    sro    = simres,
    ylog   = FALSE)
pev$fig

}
```

`plot_sr_tc` *Plots Timecourse of Rules Simulations*

Description

Plots the timecourse of `simulate_rules()` output.

Usage

```
plot_sr_tc(
  sro = NULL,
  dvcols = NULL,
  fpage = 1,
  fcol = "id",
  xcol = "time",
  error_msgs = NULL,
  ylog = TRUE,
  ylab_str = "Output",
  xlab_str = "Time",
  post_proc = "fig  = fig + ggplot2::theme_light()",
  fncol = 4,
  fnrow = 2
)
```

Arguments

`sro`	Output of 'simulate_rules()'.
`dvcols`	Character vector of dependent variables.
`fpage`	If facets are selected and multiple pages are generated then this indcates the page to return.
`fcol`	Name of column to facet by or `NULL` to disable faceting (`"id"`).
`xcol`	Name of column to take x-data from (`"time"`).
`error_msgs`	Named list with error messages to overwrite (`NULL`
`ylog`	Boolean to enable log10 scaling of the y-axis (`TRUE`
`ylab_str`	Label for the y-axis (`"Output"`
`xlab_str`	Label for the x-axis (`"Output"`
`post_proc`	Character object with post processing post-processing code for the figure object named `fig` internall (`"fig = fig + theme_light()"`)
`fncol`	Number of columns in faceted output.
`fnrow`	Number of rows in faceted output.

Details

For a detailed examples see `vignette("clinical_trial_simulation", package = "ruminate")`.

Value

List with the followin1g elements:

- isgood: Return status of the function.
- msgs: Error or warning messages if any issues were encountered.
- npages: Total number of pages using the current configuration.
- error_msgs: List of error messages used.
- dsp: Intermediate dataset generated from `sro` to plot in ggplot.
- fig: Figure generated.

Examples

```
library(formods)
library(ggplot2)

# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html

# None of this will work if rxode2 isn't installed:
if(is_installed("rxode2")){
library(rxode2)
set.seed(8675309)
rxSetSeed(8675309)
```

```
my_model = function ()
{
   description <- "One compartment PK model with linear clearance using differential equations"
    ini({
        lka <- 0.45
        label("Absorption rate (Ka)")
        lcl <- 1
        label("Clearance (CL)")
        lvc <- 3.45
        label("Central volume of distribution (V)")
        propSd <- c(0, 0.5)
        label("Proportional residual error (fraction)")
        etalcl ~ 0.1
    })
    model({
        ka <- exp(lka)
        cl <- exp(lcl + etalcl)
        vc <- exp(lvc)
        kel <- cl/vc
        d/dt(depot) <- -ka * depot
        d/dt(central) <- ka * depot - kel * central
        Cc <- central/vc
        Cc ~ prop(propSd)
    })
}

# This creates an rxode2 object
object  = rxode(my_model)

# If you want details about the parameters, states, etc
# in the model you can use this:
rxdetails = fetch_rxinfo(object)

rxdetails$elements

# Next we will create subjects. To do that we need to
# specify information about covariates:
nsub = 2
covs = list(
  WT         = list(type     = "continuous",
                    sampling = "log-normal",
                    values   = c(70, .15))
)

subs = mk_subjects(object = object,
                   nsub   = nsub,
                   covs   = covs)

head(subs$subjects)

rules = list(
  dose = list(
    condition = "TRUE",
```

```
    action    = list(
      type  = "dose",
      state     = "central",
      values    = "c(1)",
      times     = "c(0)",
      durations = "c(0)")
    )
)

# We evaulate the rules for dosing at time 0
eval_times =  0

# Stop 2 months after the last dose
output_times = seq(0, 56, 1)

# This runs the rule-based simulations
simres =
  simulate_rules(
    object        = object,
    subjects      = subs[["subjects"]],
    eval_times    = eval_times,
    output_times  = output_times,
    rules         = rules)

# First subject data:
sub_1 = simres$simall[simres$simall$id == 1, ]

# First subjects events
evall = as.data.frame(simres$evall)
ev_sub_1 = evall[evall$id ==1, ]

# All of the simulation data
simall = simres$simall
simall$id = as.factor(simall$id)

# Timecourse
psim =
  plot_sr_tc(
    sro    = simres,
    dvcols = "Cc")
psim$fig

# Events
pev =
  plot_sr_ev(
    sro    = simres,
    ylog   = FALSE)
pev$fig

}
```

`ruminate` *ruminate: Shiny app and module to facilitate pharamacometrics analysis*

Description

This is done by creating a Shiny interface to different tools for data transformation (`dplyr` and `tidyr`), plotting (`ggplot2`), and noncompartmental analysis (`PKNCA`). These results can be reported in Excel, Word or PowerPoint. The state of the app can be saved and loaded at a later date. When saved, a script is generated to reproduce the different actions in the Shiny interface.

Runs the pharmacometrics ruminate app.

Usage

```
ruminate(
  host = "127.0.0.1",
  port = 3838,
  server_opts = list(shiny.maxRequestSize = 30 * 1024^2),
  devmode = FALSE,
  mksession = FALSE
)
```

Arguments

`host`	Hostname of the server ("127.0.0.1")
`port`	Port number for the app (3838)
`server_opts`	List of options (names) and their vlues (value) e.g. `list(shiny.maxRequestSize = 30 * 1024^2)`.
`devmode`	Boolean value, when TRUE will run ruminate with development modules.
`mksession`	Boolean value, when TRUE will load test session data for app testing.

Value

Nothing is returned, this function just runs the built-in ruminate app.

Author(s)

Maintainer: John Harrold <`john.m.harrold@gmail.com`> (ORCID)

See Also

`https://ruminate.ubiquity.tools/`

Examples

```
if (interactive()) {
ruminate()
}
```

`ruminate_check` *Checks* `ruminate` *Dependencies*

Description

Looks at the suggested dependencies and checks to make sure they are installed.

Usage

```
ruminate_check(verbose = TRUE)
```

Arguments

`verbose`	Logical indicating if messages should be displayed

Value

List with the following elements:

- all_found: Boolean indicating if all packages were found
- found_pkgs: Character vector of found packages
- missing_pkgs: Character vector of missing packages

Examples

```
fcres =ruminate_check()
```

`run_nca_components` *Runs NCA for the Current Analysis*

Description

Takes the current state and runs the current analysis in that state.

Usage

```
run_nca_components(
  state,
  components = c("nca", "fg_ind_obs", "tb_ind_obs", "tb_ind_params"),
  verbose = TRUE
)
```

Arguments

`state`	NCA state from `NCA_fetch_state()`
`components`	List of components to run. By default it will run all of the following. If you just need to regenerate a figure based on the current nca results you can just specify that component. These are the valid
`verbose`	Logical to enable or disable messaging components: • nca: Run NCA analysis • fg_ind_obs: Build the figure(s) with the indiviudal observations. • tb_ind_obs: Build the table(s) with the indiviudal observations. • tb_ind_params: Build the table(s) with the indiviudal parameters.

Value

List with the following components:

- isgood: Return status of the function.
- msgs: Error messages if any issues were encountered.
- nca_res: PKNCA results if run was successful.

Examples

```
# We need a state object to use below
sess_res = NCA_test_mksession()
state = sess_res$state

state = run_nca_components(state, components="tb_ind_params")
```

`rx2other` *Converts an rxode2 Object Into Specified Model Format*

Description

If you have an rxode2 or nlmixr2 model object you can use this function to translate that object into other formats. See output_type below for the allowed formats.

In order to do this you need at least one between-subject variability term and one endpoint. If these are missing then dummy values will be added. The dummy values for between-subject variablitiy are IIV will be `POP_RUMINATE`, `TV_RUMINATE`, and `ETA.RUMINATE`. The dummy terms for endpoints are `OUT_RUMINATE` and `add.OUT_RUMINATE`.

Usage

```
rx2other(
  object,
  out_type = "nonmem",
  dataset = NULL,
  export_name = "my_model",
  export_path = tempfile(pattern = "dir")
)
```

Arguments

`object`	rxode2 model object
`out_type`	Output type (either "nonmem", "monolix")
`dataset`	Optional dataset
`export_name`	Basename for models used
`export_path`	Location to place output files (default `tempdir()`)

Details

Known issues: If you have specified bioavailability in the model, it will fail on the Monolix conversion.

Value

List with the following elements:

- isgood: Return status of the function.
- msgs: Error or warning messages if any issues were encountered.
- files: If succesful this will contain a list with an entry for each file generated to support the requested format. the current file format. For example if "nonmem" was selected this will include elements for "ctl" and "csv". Each of these are lists with the following format:
  - fn: Exported file name
  - fn_full: Exported file name with the full path.
  - contents: Contents of the file.

Examples

```
library(ruminate)
if( Sys.getenv("ruminate_rxfamily_found") == "TRUE"){
 # First create an rxode2 model:
 library(rxode2)
 one.compartment <- function() {
   rxode2::ini({
     tka <- log(1.57); label("Ka")
     tcl <- log(2.72); label("Cl")
     tv <- log(31.5); label("V")
     eta.ka ~ 0.6
     eta.cl ~ 0.3
     eta.v ~ 0.1
     add.sd <- 0.7
   })
   # and a model block with the error specification and model specification
   rxode2::model({
     ka <- exp(tka + eta.ka)
     cl <- exp(tcl + eta.cl)
     v <- exp(tv + eta.v)
     d/dt(depot) <- -ka * depot
     d/dt(center) <- ka * depot - cl / v * center
```

```
      cp <- center / v
      cp ~ add(add.sd)
    })
  }

  nmout = rx2other(one.compartment, out_type="nonmem")

}
```

`simulate_rules` *Rule-Based simulates*

Description

Simulate an rxode2 model based on rules evaluated at specified time-points. For example if you want to titrate dosing based on individual plasma levels you could create a rule that changes dosing at specified time points based on the last observation of the user.

Usage

```
simulate_rules(
  object,
  subjects,
  eval_times,
  output_times,
  time_scales = NULL,
  rules,
  rx_options = list(),
  preamble = "",
  pbm = "Evaluation times",
  smooth_sampling = TRUE
)
```

Arguments

`object`	rxode2 model object An ID string that corresponds with the ID used to call the modules UI elements
`subjects`	Dataframe of subject level information.
`eval_times`	Vector of simulation times to evaulate the rules (units are time units of the system).
`output_times`	Specific output times to include. Other times will be included as well to ensure proper evaluation of the rules.
`time_scales`	Optional list with timescale information to include in the output.
`rules`	List of rules, see below for a description of the format.
`rx_options`	List of options to pass through to `rxSolve()`.

`preamble`	Character string of user-defined code to execute in rule-evaluation environment (e.g. you can put user-defined functions here).
`pbm`	Progress bar message, set to NULL to disable.
`smooth_sampling`	Boolean when TRUE will insert sampling just before dosing to make sampling smooth.

Details

For a detailed examples see `vignette("clinical_trial_simulation", package = "ruminate")`

The underlying simulations are run using `rxode2`, and as such we need an `rxode2` system object. From that we can either simulate subjects or load them from a file. Next we need to define a set of rules. These will be a set of conditions and actions. At each evaluation time point the conditions are evaluated. When a condition is met the actions associated with that condition are executed. For example, if during a visit (an evaluation time point) the trough PK is below a certain level (condition) we may want to increase the dosing regimen for the next dosing cycle (action).

Creating subjects:

Subjects are expected in a data frame with the following column headers:

- `id` Individual subject id
- Names of parameters and iiv as specified in the `ini` section of the `rxode2` function specification
- Names of covariates used in the model.

`mk_subjects()` — Creates subjects for simulation by sampling based on between-subject variability and generating covariate information based on user specifications.

Covariates:

The `covs` input is a list with the following structure:

- type: Can be either “fixed”, “discrete”, or “continuous”.
- sampling: This field is only needed for a “continuous” covariate ’ type and can be either “random”, “normal” or “log-normal”.
- values: This field depends on the type and optional sampling above.
  - fixed: A single value.
  - discrete: A vector of possible discrete elements.
  - continuous, random: Two values the first is the lower bound and the second is the upper bound.
  - continuous, normal: Two values the first is the mean and the second is the variance.
  - continuous, log-normal: Two values the first is the mean and the second is the variance.

This examples shows the `SEX_ID` randomly sampled from the values specified, `SUBTYPE_ID` fixed at a value, and `WT` sampled from a log-normal distribution.

```
covs = list(
  SEX_ID     = list(type     = "discrete",
                    values   = c(0,1)),
  SUBTYPE_ID = list(type     = "fixed",
                    values   = c(0)),
  WT         = list(type     = "continuous",
```

```
                          sampling = "log-normal",
                          values   = c(70, .15))
)
```

Rule-based simulations:

`simulate_rules()` — This will run simulations based on the rule definitions below.

Rules:

Rules are a named list where the list name can be a short descriptive label used to remember what the rule does. These names will be returned as columns in the simulated data frame.

- `condition`: Character string that evaluates to either `TRUE` or `FALSE`. When true the `action` portion will be triggered. For a list of objects available see the Rule-evaluation environment below.
- `fail_flag`: Flag set in the `rule_id` column when the condition is not met (set to `"false"` if not specified).
- `true_flag`: Flag set in the `rule_id` column when the condition is met (set to `"true"` if not specified).
- `action`: This is what the rule will trigger can be any of the following:
  - `type`: This defines the action type and can be either `"dose"`, `"set state"`, or `"manual"`.

Based on the `type` the `action` field will expect different elements.

Dosing:

- `action`
  - `type`: `"dose"`
  - `values`: Character string that evaluates as a numeric vector dosing amounts (e.g. `"c(3, 3, 3, 3)"`)
  - `times`: Character string that evaluates as a numeric vector of times (e.g. `"c(0, 14, 28, 42)"`)
  - `durations`: Character string that evaluates as a numeric vector of durations (e.g. `"c(0, 0, 0, 0)"`, zero for bolus dosing)

Changing a state value:

- `action`
  - `type`: `"set state"`
  - `state`: Character string with the name of the state to set (`"Ac"`)
  - `value`: Character string that evaluates as a numeric value for state (e.g. `"Ac/2"` would set the state to half the value of Ac at the evaluation point)

Manual modification of the simulation:

- `action`
  - `type`: `"manual"`
  - `code`: Character string of code to evaluate.

Rule-evaluation environment:

Beyond simple simulations it will be necessary to execute actions based on the current or previous state of the system. For this reason, when a `condition` or elements of the action (e.g., the `values`, `times` and `durations` of a dose action type) are being evaluated, the following objects will be available at each evaluation point:

- outputs: The value of each model output.
- states: The value of each named state or compartment.

- covariates: The value of each named covariate.
- subject-level parameters: The value of each named parameter.
- rule value: The last value the rule evaluated as.
- `id`: Current subject id.
- `time`: Current evaluation time.
- `SI_SUB_HISTORY`: A data frame of the simulation history of the current subject up to the current evaluation point.
- `SI_subjects`: The subjects data frame.
- `SI_eval_times`: Vector of the evaluation times.
- `SI_interval_ev`: The events table in it's current state for the given simulation interval.
- `SI_ev_history`: This is the history of the event table containing all the events leading up to the current interval.
- `SI_ud_history`: This is a free form object the user can define or alter within the "manual"action type (ud-user defined, history).

The following functions will be available::

- `SI_fpd`: This function will fetch the previous dose (fpd) for the given id and state. For example for the current `id` and the state `Ac` you would do the following:

```
SI_fpd(id=id, state="Ac")
```

Time scales:
You can include columns in your output for different time scales if you wish. You need to create a list in the format below. One element should be `system` with a short name for the system time scale. The next should be `details` which is a list containing short names for each time scale you want to include. Each of these is a list with a verbose name for the time scale (`verb`) and a numerical conversion indicating how that time scale relates to the others. Here we define weeks and days on the basis of seconds.

```
time_scales = list(system="days",
                details= list(
                  weeks = list(verb="Weeks",    conv=1/(60*60*24*7)),
                  days  = list(verb="Days",     conv=1/(60*60*24))))
```

Value

List with the following elements:

- isgood: Return status of the function.
- msgs: Error or warning messages if any issues were encountered.
- simall: Simulation results.
- ev_history: The event table for the entire simulation history.
- eval_times: Evaluation time points at the system time scale
- eval_times_df: Data frame of the evaluation time points with a column for the system time scale and then columns for named time scales.

Examples

```
library(formods)
library(ggplot2)

# For more information see the Clinical Trial Simulation vignette:
# https://ruminate.ubiquity.tools/articles/clinical_trial_simulation.html

# None of this will work if rxode2 isn't installed:
if(is_installed("rxode2")){
library(rxode2)
set.seed(8675309)
rxSetSeed(8675309)

my_model = function ()
{
   description <- "One compartment PK model with linear clearance using differential equations"
    ini({
        lka <- 0.45
        label("Absorption rate (Ka)")
        lcl <- 1
        label("Clearance (CL)")
        lvc <- 3.45
        label("Central volume of distribution (V)")
        propSd <- c(0, 0.5)
        label("Proportional residual error (fraction)")
        etalcl ~ 0.1
    })
    model({
        ka <- exp(lka)
        cl <- exp(lcl + etalcl)
        vc <- exp(lvc)
        kel <- cl/vc
        d/dt(depot) <- -ka * depot
        d/dt(central) <- ka * depot - kel * central
        Cc <- central/vc
        Cc ~ prop(propSd)
    })
}

# This creates an rxode2 object
object  = rxode(my_model)

# If you want details about the parameters, states, etc
# in the model you can use this:
rxdetails = fetch_rxinfo(object)

rxdetails$elements

# Next we will create subjects. To do that we need to
# specify information about covariates:
nsub = 2
covs = list(
```

```
  WT         = list(type     = "continuous",
                    sampling = "log-normal",
                    values   = c(70, .15))
)

subs = mk_subjects(object = object,
                   nsub   = nsub,
                   covs   = covs)

head(subs$subjects)

rules = list(
  dose = list(
    condition = "TRUE",
    action    = list(
      type  = "dose",
      state     = "central",
      values    = "c(1)",
      times     = "c(0)",
      durations = "c(0)")
    )
)

# We evaulate the rules for dosing at time 0
eval_times =  0

# Stop 2 months after the last dose
output_times = seq(0, 56, 1)

# This runs the rule-based simulations
simres =
  simulate_rules(
    object        = object,
    subjects      = subs[["subjects"]],
    eval_times    = eval_times,
    output_times  = output_times,
    rules         = rules)

# First subject data:
sub_1 = simres$simall[simres$simall$id == 1, ]

# First subjects events
evall = as.data.frame(simres$evall)
ev_sub_1 = evall[evall$id ==1, ]

# All of the simulation data
simall = simres$simall
simall$id = as.factor(simall$id)

# Timecourse
psim =
  plot_sr_tc(
    sro    = simres,
```

```
    dvcols = "Cc")
psim$fig

# Events
pev =
  plot_sr_ev(
    sro    = simres,
    ylog   = FALSE)
pev$fig

}
```

Index