bupaR

Business Process Analysis with

Getting Started

```
install.packages("bupaR")
install.packages("eventdataR")
install.packages("xesreadR")
install.packages("edeaR")
install.packages("processmapR")
install.packages("processmonitR")
```

```
library(bupaR)
```

```
devtools::install_github(
    "gertjanssenswillen/<package>"
    )
```

In order to start using bupaR, install the required packages from the Comprehensive R Archive Network (CRAN), using the `install.packages` function.

The packages can be loaded with the `library` function. Loading bupaR will load all the related packages.

The latest versions of the packages can be installed from github using the `install_github` function from devtools. Contributions and bugfixes to the packages are welcome on github. Please visit https://github.com/gertjanssenswillen/bupaR.

Process Analysis Workflows

All functions within the scope of **bupaR** are designed to be used with the `magrittr::%>%` symbol. This symbol allows the first argument of a function to be moved upfront.

`x %>% f(y)` is the same as `f(x,y)`

`y %>% f(x, ., z)` is the same as `f(x,y,z)`

Using this symbol makes it very easy to create workflows for you process analysis, as in the examples below.

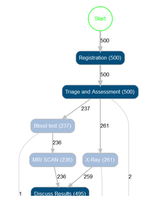
```
patients %>%
    filter_trace_length(upper = 10) %>%
    precedence_matrix() %>%
    plot()
```

```
patients %>%
    sample_n(size = 100) %>%
    filter_resource_frequency(perc = 0.8) %>%
    resource_frequency("resource") %>%
    plot()
```

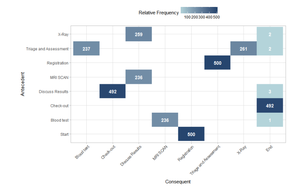
Process Visualizations

The **processmapR** package provides functions to visualize processes, both from a control-flow perspective and from a resource perspective. The `process_map` function allows the user to analyse control-flow from a frequency and a performance perspectives. The `precendence_matrix` provides a more compact overview of the process flows. Furthermore, the package provides functions to explore (in)frequent traces and a dotted chart (including an interactive version). Also resource maps and matrices can be made.

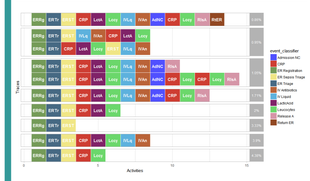
`processmapR::process_map`



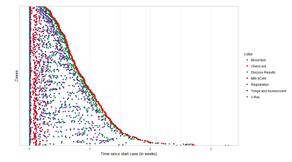
`processmapR::precedence_matrix %>% plot`



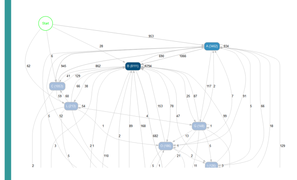
`processmapR::trace_explorer`



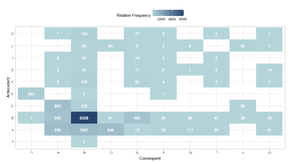
`processmapR::idotted_chart`

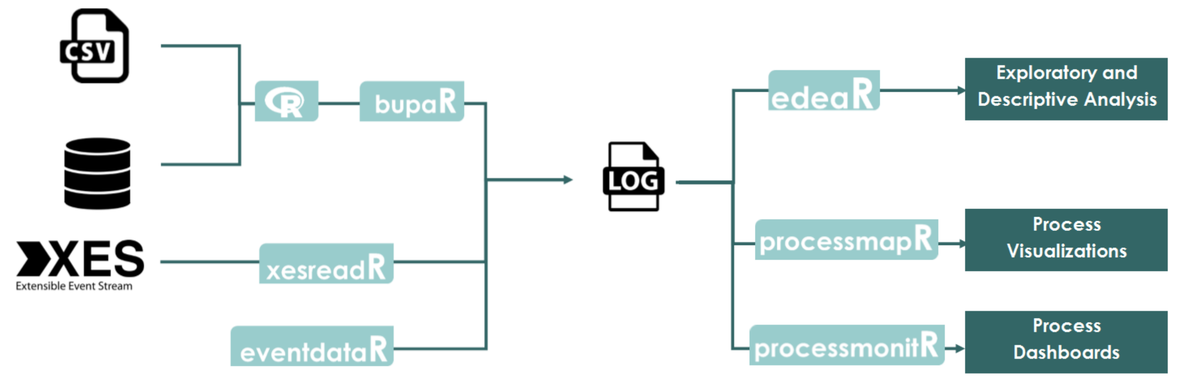


`processmapR::resource_map`



`processmapR::resource_matrix %>% plot`





Over the past decades, the open source statistical language R has seen an enormous increase in popularity, not only among data science researchers, but also within companies. One of the reasons for this rising popularity is the R-package ecosystem on CRAN and github to which everyone can contribute. Recently, the number of packages available on CRAN has exceeded 10.000. These provide a huge range of functionalities, covering a diverse set of techniques and applications.

bupaR is an open-source suite for the handling and analysis of business process data in R developed by the Business Informatics research group at Hasselt University, Belgium. The central package includes basic functionality for creating event log objects in R. It contains several functions to get information about an event log and also provides specific event log versions of generic R functions.

edeaR, which stands for Exploratory and Descriptive Event-Data Analysis, provides several functions for in-depth analysis of event logs, as well as a diverse set of subsetting methods. Process visualizations can be made with the **processmapR** package, while the **processmonitR** package provides various off-the-shelf dashboards for process monitoring. An interface with the XES-standard is provided by the **xesreadR** package, while a set of event logs are supplied through the **eventdataR** package.

Create Event Log Objects

```
bupaR::eventlog(eventlog = data,
    case_id = "order_number",
    activity_id = "activity",
    activity_instance_id = "activity_nr",
    timestamp = "time",
    lifecycle_id = "status",
    resource_id = "originator")
```

```
bupaR::simple_eventlog(eventlog = data,
    case_id = "order_number",
    activity_id = "activity",
    timestamp = "time")
```

```
bupaR::ieventlog(eventlog = data)
bupaR::isimple_eventlog(eventlog = data)
```

An **eventlog**-object can be created using the **eventlog** function. This function needs as arguments a data.frame and the column names of the appropriate fields describing the case identifier, activity identifier, timestamp, lifecycle transition, resource and an activity instance identifier.

An event log with minimal requirements (timestamp, case and activity identifier) can be created with the **simple_eventlog** function.

Both functions have an alternative, prefixed with the letter *i* for *interface*, in order to configure the identifiers with a GUI. In that case only the data needs to be provided as argument.

Basic Event Log Functionalities

```
bupaR::summary(eventlog)
bupaR::activities(eventlog)
bupaR::cases(eventlog)
bupaR::resources(eventlog)
bupaR::traces(eventlog)
```

Create a summary of an **eventlog**-object. Retrieve basic information on activities, cases, resources and traces, such as the absolute and relative frequences.

```
bupaR::mapping(eventlog)
bupaR::activity_id(eventlog)
bupaR::activity_instance_id(eventlog)
bupaR::case_id(eventlog)
bupaR::lifecycle_id(eventlog)
bupaR::resource_id(eventlog)
bupaR::timestamp(eventlog)
```

Obtain the mapping of an eventlog (set of identifiers) or obtain single identifiers.

```
bupaR::n_activities(eventlog)
bupaR::n_activity_instances(eventlog)
bupaR::n_cases(eventlog)
bupaR::n_events(eventlog)
bupaR::n_resource(eventlog)
bupaR::n_traces(eventlog)
```

Calculate the number of distinct activities, activity instances, cases, events, resources and traces.

```
bupaR::slice(eventlog, n:m)
```

Subset of slice of the event log from row n until row m.[1]

```
bupaR::sample_n(eventlog, n)
```

Sample n cases from the event log.[1]

```
bupaR::mutate(eventlog, ...)
```

Add new variables to the event log (using arithmetic operations, case_when, ifelse, etc.).[1]

```
bupaR::group_by(eventlog, ...)
```

Group an event log on one or more event or case attributes.[1]

[1]These functions are eventlog-adaptations from the same-titled dplyr-functions. For more information, check dplyr.tidyverse.org.

Reading and Writing XES-files

```
xesreadR::read_xes(xesfile)
xesreadR::read_xes_cases(xesfile)
xesreadR::write_xes(eventlog,
    xesfile,
    case_attributes)
```

Event logs and a list of cases with attributes can be imported from a XES-file using the `read_xes` and **`read_xes_cases`**. Note that the `read_xes` also returns the case_attributes in the event log. `write_xes` can be used to write an event log to a XES-file.

Event Data Repository

```
eventdataR::BPIC_14_incident_log
eventdataR::BPIC_14_incident_case_attributes
eventdataR::BPIC_15_1
eventdataR::sepsis
eventdataR::patients
```

The **eventdataR** package gives access to several event log examples, both real-life and artificial. The nature and origin of each of the data sets can be found in the accompanying documentation.

Created and maintained by Gert Janssenswillen

Email gert.janssenswillen@uhasselt.be

Contributors benoit.depaire@uhasselt.be

marijke.swennen@uhasselt.be

mieke.jans@uhasselt.be

Website www.bupar.net

Exploratory and Descriptive Event Data Analysis

```
edeaR::activity_frequency
edeaR::activity_presence
edeaR::end_activities
edeaR::idle_time
edeaR::number_of_repetitions
edeaR::number_of_selfloops
edeaR::number_of_traces
edeaR::processing_time
edeaR::resource_frequency
edeaR::resource_involvement
edeaR::resource_specialisation
edeaR::size_of_repetitions
edeaR::size_of_selfloops
edeaR::start_activities
edeaR::throughput_time
edeaR::trace_coverage
edeaR::trace_length
```

edeaR provides a varied set of metrics to analyse event logs. Each of the metrics, listed in the table to the left, has a set of analysis-levels at which they can be computed. These levels can be log, case, trace, activity, resource and resource-activity. These levels allow the user to conduct the analysis at the appropriate level of granularity.

A metric can be calculated as follows.

```
eventlog %>%
    <metric>(
        level= "log",
        ...)
```

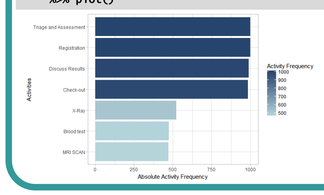
1. The dataset
2. The metric to compute
3. The desired analysis level
4. Optional arguments, e.g. time units in case of throughput time.

```
patients %>% activity_frequency(level = "activity")
```

```
# A tibble: 7 x 3
             handling absolute   relative
               <chr>    <int>      <dbl>
1          Blood test      474 0.08710033
2           Check-out      984 0.18081588
3     Discuss Results      990 0.18191841
4            MRI SCAN      472 0.08673282
5        Registration     1000 0.18375597
6 Triage and Assessment    1000 0.18375597
```

While each of the metrics returns a data frame or vector with numerical results, they can be very easily visualized using the generic `plot` function. The latter has been adapted to accomodate each of the metrics and analysis levels with an appropriate visual, and can be added to the workflow with just another %>% symbol, as shown in the example on the left.

```
patients %>% activity_frequency(level = "activity")
    %>% plot()
```



Although no prior knowledge about visualization methods in R is necessary, the resulting plot is a ggplot-object which can be customized according to the wishes of the end users when desired. By adding ggplot-layers after the plot function, one can change the colors of the plot, the labels on the axis, add titles or adjust the overal theme. More information on ggplot2 can be found on http://ggplot2.org/.

Event Data Subsetting

EVENT FILTERS

```
edeaR::ifilter_activity
edeaR::ifilter_activity_frequency
edeaR::ifilter_attributes
edeaR::ifilter_resource
edeaR::ifilter_resource_frequency
edeaR::ifilter_time_period
edeaR::ifilter_trim
```

CASE FILTERS

```
edeaR::ifilter_activity_presence
edeaR::ifilter_case
edeaR::ifilter_endpoints
edeaR::ifilter_precedence
edeaR::ifilter_processing_time
edeaR::ifilter_throughput_time
edeaR::ifilter_time_period
edeaR::ifilter_trace_frequency
edeaR::ifilter_trace_length
```

Subsetting event data with edeaR can be done on two levels: on the level of events or on the level of cases.

Each of the subsetting methods starts with the word *filter*, and each of them has an interactive alternative starting with *ifilter*. Most of the filters can be specified in various ways by using different arguments, and each of the filters has a reverse argument to reverse the selection.

Filtering events can be done by activity label, activity frequency, attribute conditions, resource labels, by trimming cases to a time period, or by trimming cases between specified activities.

Filtering cases can be done based on the presence or absence of activities, based on case identifier, the end points of the case (i.e. start and end activity), precedence conditions, processing or throughput time, time periods, trace freqency or trace length.

```
patients %>% filter_activity_frequency(percentile = 0.8)
patients %>% filter_activity_presence(
    c("Registration","Check-out"), method = "all")
patients %>% filter_time_period(
    start_point = ymd(20170101),
    end_point = ymd(20170131),
    filter_method = "start", reverse = T)
```

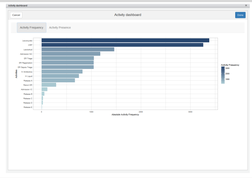
The examples filter the log based on (1) frequent activities covering 80% of the events, (2) cases having both Registration and Check-out activities and (3) cases which did **not** started in January 2017.

Process Dashboards

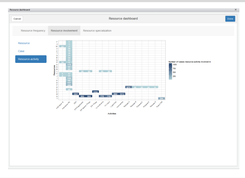
The **processmonitR** package provides predefined dashboards to interactively monitor processes from different perspectives. Currently, four different dashboards are provided: 1) an activity dashboard, focused on activities, 2) a resource dashboard, focusing on resources, 3) a rework dashboard, focusing on rework and waste, such as self-loops and repetitions, and a 4) performance dashboard, focusing on the time perspective, i.e. throughput time, processing time and idle time.

Each dashboard combines several of the metrics and visualization from other **bupaR** packages into easy to use and navigate dashboards. The dashboards, implemented in Shiny, can be used as standalone dashboards, or incorporated into larger, tailor-made process monitoring dashboards.

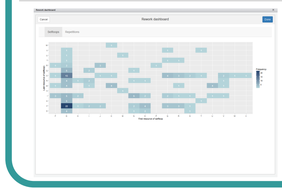
`processmonitR::activity_dashboard`



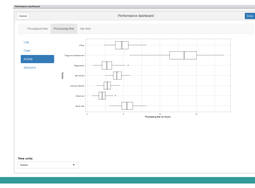
`processmonitR::resource_dashboard`



`processmonitR::rework_dashboard`



`processmonitR::performance_dashboard`



Conditional Process Analysis

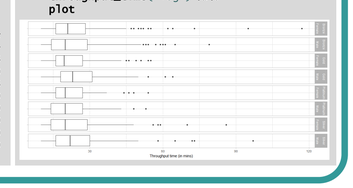
bupaR facilitates conditional process analysis, by making it possible to compute metrics and visualizations for values of case/event attributes or even combination of attributes.

This is achieved by an adaption of the `group_by` function from dplyr for eventlogs. In order to condition a certain metric on one or more variables, the group_by function needs to be inserted as shown in the examples on the right. Also the visualizations, as created with `plot`, will reflect the distinction between different groups.

```
eventlog %>%
    group_by(label, gender) %>%
    throughput_time("log")
```

```
# A tibble: 8 x 10
     label gender   min    q1 median  mean    q3   max st_dev   iqr
     <fctr> <fctr> <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>
1   Bronze Female    10 16.00   21.0 24.98 28.25   117  14.35 12.25
2   Silver Female    10 14.00   20.0 23.03 29.00    86  11.93 15.00
3   Bronze   Male    10 14.25   20.0 23.84 29.00    79  12.26 14.75
4     Gold Female    10 16.00   19.5 22.81 27.00    55  11.08 11.00
5   Silver   Male    10 16.00   22.0 25.69 30.00    97  14.53 14.00
6     Gold   Male    10 18.00   23.0 25.62 31.00    64  11.07 13.00
7 Platinum   Male    10 14.00   20.0 22.60 27.00    53  10.45 13.00
8 Platinum Female    10 16.00   20.0 22.51 27.00    54   9.73 11.00
```

```
eventlog %>%
    group_by(label, gender) %>%
    throughput_time("log") %>%
    plot
```



This poster discusses the bupaR packages according to the following released versions:

bupaR 0.3.0
edeaR 0.7.0
eventdataR 0.1.1
processmapR 0.2.0
processmonitR 0.1.0
xesreadR 0.2.1