Extensions to the bupaR ecosystem: an overview

Tutorial

Getting started

Installing required packages

The following packages can be installed from CRAN using `install.packages`. Note that other core packages of bupaR, such as edeaR can be installed this way as well. However, we only install the packages required for this tutorial.

```
install.packages("bupaR")
install.packages("processmapR")
install.packages("daqapo")
install.packages("processanimateR")
install.packages("heuristicsmineR")
```

The packages logbuildR, collaborateR and propro can be installed from github, using the `remotes` installation function.

```
install.packages("remotes")
remotes::install_github("bupaverse/logbuildR")
remotes::install_github("bupaverse/collaborateR")
remotes::install_github("bupaverse/propro")
```

Load packages

Using the `library` function, we can load the packages to use them.

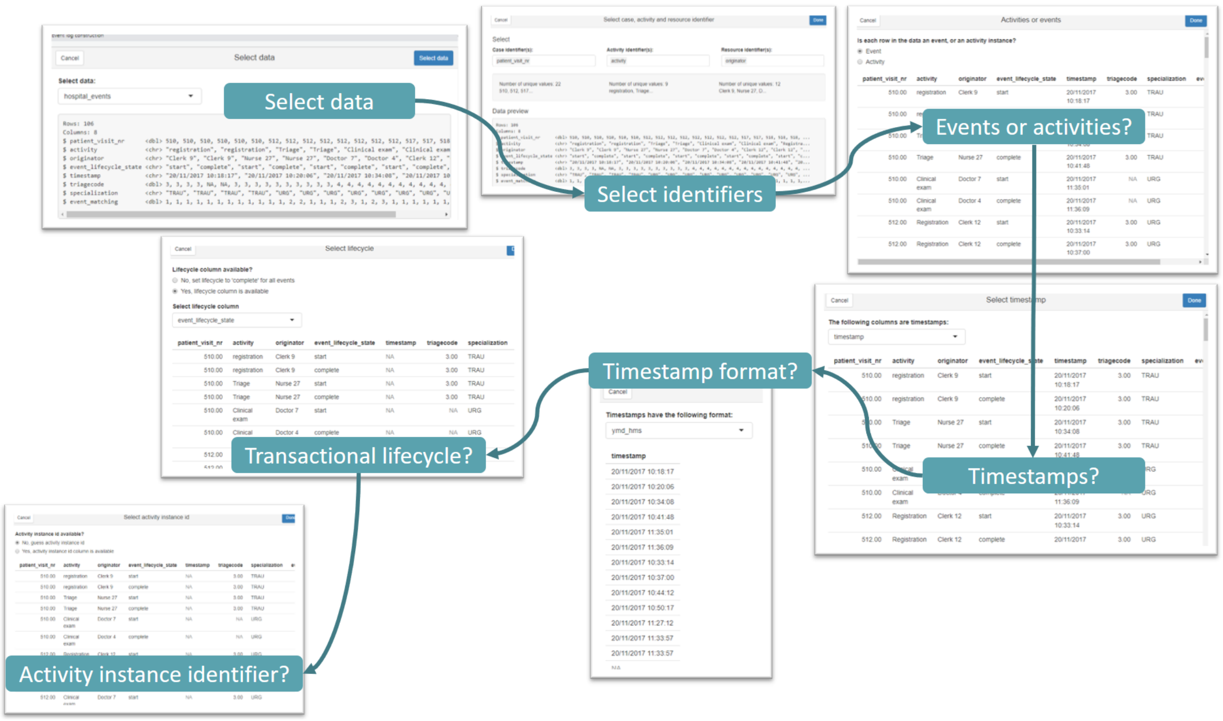
```
library(bupaR)
library(logbuildR)
library(daqapo)
library(heuristicsmineR)
library(propro)
library(processanimateR)
library(collaborateR)
```

LogbuildR

Starting from a regular R data.frame, logbuildR helps the user to go through different steps in order to prepare the data for process analytics. To start the process, you can execute the function `build_log()`.

```r
build_log()
```

This will open a dialogue window that will ask you to select your data. In subsequent steps, you will be able to prepare the data by answering the questions prompted. The Figure below shows an example of the different steps that the logbuildR goes through. Of course, the specific sequence of questions and steps can be different, depending on the format of the input data.



An example of the end-to-end logbuildR workflow will be given in the accompanying screen-cast.

DaQAPO

The daqapo packages comes with an example dataset from a hospital process, that can be used to illustrate different data quality issues.

```r
data("hospital")
hospital
```

```
## # A tibble: 53 x 7
##    patient_visit_nr activity originator start_ts complete_ts triagecode
##               <dbl> <chr>    <chr>      <chr>    <chr>            <dbl>
##  1              510 registr~ Clerk 9    20/11/2~ 20/11/2017~          3
##  2              512 Registr~ Clerk 12   20/11/2~ 20/11/2017~          3
##  3              510 Triage   Nurse 27   20/11/2~ 20/11/2017~          3
##  4              512 Triage   Nurse 27   20/11/2~ 20/11/2017~          3
##  5              512 Clinica~ Doctor 7   20/11/2~ 20/11/2017~          3
##  6              510 Clinica~ Doctor 7   20/11/2~ 20/11/2017~         NA
##  7              517 Triage   Nurse 17   21/11/2~ 21/11/2017~          3
##  8              518 Registr~ Clerk 12   21/11/2~ 21/11/2017~          4
## 9               518 Registr~ Clerk 6    21/11/2~ 21/11/2017~          4
## 10              518 Registr~ Clerk 9    21/11/2~ 21/11/2017~          4
## # ... with 43 more rows, and 1 more variable: specialization <chr>
```

In order use this dataset, we convert it to an activity log, using the code below. This code has been put together using the logbuildR. It gives appropriate names to the timestamp columns, converts them to actual date-objects, and constructs and activity log.

```
data("hospital")
hospital %>%
  rename(start = start_ts,
         complete = complete_ts) %>%
  convert_timestamps(c("start","complete"), format = dmy_hms) %>%
  activitylog(case_id = "patient_visit_nr",
              activity_id = "activity",
              resource_id = "originator",
              lifecycle_ids = c("start", "complete")) -> hospital
hospital
```

```
## # A tibble: 53 x 7
##    patient_visit_nr activity originator start               complete
##               <dbl> <chr>    <chr>      <dttm>              <dttm>
##  1              510 registr~ Clerk 9    2017-11-20 10:18:17 2017-11-20 10:20:06
##  2              512 Registr~ Clerk 12   2017-11-20 10:33:14 2017-11-20 10:37:00
##  3              510 Triage   Nurse 27   2017-11-20 10:34:08 2017-11-20 10:41:48
##  4              512 Triage   Nurse 27   2017-11-20 10:44:12 2017-11-20 10:50:17
##  5              512 Clinica~ Doctor 7   2017-11-20 11:27:12 2017-11-20 11:33:57
##  6              510 Clinica~ Doctor 7   2017-11-20 11:35:01 2017-11-20 11:36:09
##  7              517 Triage   Nurse 17   2017-11-21 11:35:16 2017-11-21 11:39:00
##  8              518 Registr~ Clerk 12   2017-11-21 11:45:16 2017-11-21 11:22:16
##  9              518 Registr~ Clerk 6    2017-11-21 11:45:16 2017-11-21 11:22:16
## 10              518 Registr~ Clerk 9    2017-11-21 11:45:16 2017-11-21 11:22:16
## # ... with 43 more rows, and 2 more variables: triagecode <dbl>,
## #   specialization <chr>
```

In order to easily find the quality assessment functionalities, all function starts with the prefix "detect_". Below, we will show some examples.

Detect Activity Frequency Violations

In the situation that the frequencies of activities within a case should have a certain fixed number, the `detect_activity_frequency_violations` function can be used. You can provide it with one or more (activity, frequency) pairs, and it will check whether any cases violate this condition.

```
hospital %>%
  detect_activity_frequency_violations("Registration" = 1,
                                       "Clinical exam" = 1)
```

```
## *** OUTPUT ***
```

```
## For 3 cases in the activity log (13.6363636363636%) an anomaly is detected.
## The anomalies are spread over the following cases:

## # A tibble: 3 x 3
##   patient_visit_nr activity          n
##              <dbl> <chr>         <int>
## 1              518 Registration      3
## 2              512 Clinical exam     2
## 3              535 Registration      2
```

Here, we can see that two cases contain more than 1 registration, while one contains more than 1 clinical exam.

Detect Case ID Sequence Gaps

In event logs where the case identifiers are incremental numerical values, it is useful to check whether any gaps exist in the numerical sequence, which can be an indication of missing data. This can be done with the `detect_case_id_sequence_gaps` function. Below, we see that there are no case id's with values 511, and 513 to 516

```
hospital %>%
  detect_case_id_sequence_gaps()
```

```
## *** OUTPUT ***

## It was checked whether there are gaps in the sequence of case IDs

## From the 27 expected cases in the activity log, ranging from 510 to 536, 5 (18.52%) are missing.

## These case numbers are:

##   case present
## 1  511   FALSE
## 2  513   FALSE
## 3  514   FALSE
## 4  515   FALSE
## 5  516   FALSE
```

Detect Inactive Periods

Another approach to check for missing data is to detect inactive periods in the data. This can be done by checking for periods of times in which no cases or arriving, or periods when no activity instances have been recorded.

The table below shows 9 periods in which no new cases arrived in the process (given a threshold of 30 minutes.)

```
hospital %>%
  detect_inactive_periods(threshold = 30, type = "arrivals")
```

```
## # A tibble: 9 x 3
##   period_start        period_end          time_gap
##   <dttm>              <dttm>                 <dbl>
## 1 2017-11-20 10:20:06 2017-11-21 11:35:16   1515.
## 2 2017-11-21 11:22:16 2017-11-21 11:59:41     37.4
## 3 2017-11-21 12:05:52 2017-11-21 13:43:16     97.4
## 4 2017-11-21 14:06:09 2017-11-21 15:12:17     66.1
## 5 2017-11-21 15:18:19 2017-11-21 16:42:08     83.8
## 6 2017-11-21 17:06:10 2017-11-21 18:02:10     56
## 7 2017-11-21 18:15:04 2017-11-22 10:04:57    950.
## 8 2017-11-22 10:32:56 2017-11-22 16:30:00    357.
## 9 2017-11-22 17:00:00 2017-11-22 18:00:00     60
```

When looking for periods without any activities, 3 more inactive periods were identified, leading to a total of 12 inactive periods.

```
hospital %>%
  detect_inactive_periods(threshold = 30, type = "activities")
```

```
## # A tibble: 12 x 3
##    period_start        period_end          time_gap
##    <dttm>              <dttm>                 <dbl>
##  1 2017-11-20 10:50:17 2017-11-20 11:27:12     36.9
##  2 2017-11-20 11:36:09 2017-11-21 11:35:16   1439.
##  3 2017-11-21 11:22:16 2017-11-21 11:59:41     37.4
##  4 2017-11-21 12:05:52 2017-11-21 13:26:23     80.5
##  5 2017-11-21 15:18:04 2017-11-21 16:42:08     84.1
##  6 2017-11-21 19:00:00 2017-11-22 10:01:55    902.
##  7 2017-11-22 10:32:56 2017-11-22 11:05:32     32.6
##  8 2017-11-22 11:21:59 2017-11-22 14:28:31    187.
##  9 2017-11-22 15:25:03 2017-11-22 16:30:00     65.0
## 10 2017-11-22 17:00:00 2017-11-22 18:00:00     60
## 11 2017-11-22 18:37:00 2017-11-23 17:32:12   1375.
## 12 2017-11-23 19:01:23 2017-11-24 11:08:24    967.
```

Detect Multiregistration

Multiregistration refers to the situation in which multiple activities are recorded ad the same timestamp, either within a case, or for a single resource. Example of both, using a threshold of 10 seconds, are shown below.

```
hospital %>%
  detect_multiregistration(threshold_in_seconds = 10, level_of_aggregation = "case")
```

```
## # A tibble: 11 x 7
##    patient_visit_nr activity originator start               complete
##               <dbl> <chr>    <chr>      <dttm>              <dttm>
##  1              512 Clinica~ Doctor 7   2017-11-20 11:27:12 2017-11-20 11:33:57
##  2              512 Clinica~ Doctor 7   NA                  2017-11-20 11:33:57
##  3              518 Registr~ Clerk 12   2017-11-21 11:45:16 2017-11-21 11:22:16
##  4              518 Registr~ Clerk 6    2017-11-21 11:45:16 2017-11-21 11:22:16
##  5              518 Registr~ Clerk 9    2017-11-21 11:45:16 2017-11-21 11:22:16
##  6              527 Registr~ Clerk 6    2017-11-21 18:02:10 2017-11-21 18:04:07
##  7              527 Triage   Nurse 5    2017-11-21 18:02:11 2017-11-21 18:04:08
##  8              527 Clinica~ Doctor 4   2017-11-21 18:02:13 2017-11-21 18:04:10
##  9              536 Triage   Nurse 27   2017-11-22 15:15:39 2017-11-22 15:25:01
## 10              536 Clinica~ Doctor 1   2017-11-22 15:15:40 2017-11-22 15:25:02
## 11              536 Treatme~ Nurse 27   2017-11-22 15:15:41 2017-11-22 15:25:03
## # ... with 2 more variables: triagecode <dbl>, specialization <chr>
```

```r
hospital %>%
  detect_multiregistration(threshold_in_seconds = 10, level_of_aggregation = "resource")
```

```
## Selected level of aggregation: resource

## Selected timestamp parameter value: complete

## *** OUTPUT ***

## Multi-registration is detected for 4 of the 12 resources (33.33%). These resources are:

## Doctor 7 - Nurse 5 - Nurse 27 - NA

## For the following rows in the activity log, multi-registration is detected:

## # A tibble: 9 x 7
##   patient_visit_nr activity originator start               complete
##              <dbl> <chr>    <chr>      <dttm>              <dttm>
## 1              512 Clinica~ Doctor 7   2017-11-20 11:27:12 2017-11-20 11:33:57
## 2              512 Clinica~ Doctor 7   NA                  2017-11-20 11:33:57
## 3              524 Triage   Nurse 5    2017-11-21 17:04:03 2017-11-21 17:06:05
## 4              525 Triage   Nurse 5    2017-11-21 17:04:13 2017-11-21 17:06:08
## 5              526 Triage   Nurse 5    2017-11-21 17:04:15 2017-11-21 17:06:10
## 6              536 Triage   Nurse 27   2017-11-22 15:15:39 2017-11-22 15:25:01
## 7              536 Treatme~ Nurse 27   2017-11-22 15:15:41 2017-11-22 15:25:03
## 8              533 0        <NA>       2017-11-22 18:35:00 2017-11-22 18:37:00
## 9              534 Registr~ <NA>       2017-11-22 18:35:00 2017-11-22 18:37:00
## # ... with 2 more variables: triagecode <dbl>, specialization <chr>
```

Detect Value Range Violations

As a final example, it can be checked whether attributes adhere to a certain domain. For example, let's say the triage code should always be a numeric value between 0 and 5. This can be checked using the following command. The auxiliary functions `domain_time` and `domain_categorical` can be used to check the domain for timestamps and categorical variables, respectively.

```r
hospital %>%
  detect_value_range_violations(triagecode = domain_numeric(from = 0, to = 5)) %>%
  dplyr::select(triagecode)
```

```
## # A tibble: 7 x 1
##   triagecode
##        <dbl>
## 1         NA
## 2          7
## 3          7
## 4          7
## 5          7
## 6          7
## 7          7
```

More information and examples of daqapo can be found **here**.

HeuristicsmineR

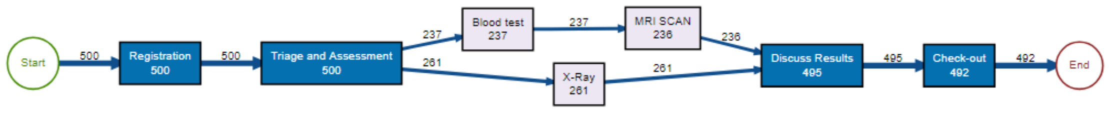
Below we show how you can use the heuristicsminer package for process discovery and subsequently for conformance checking. For this example we use the dataset `patients`.

```
patients
```

```
## Log of 5442 events consisting of:
## 7 traces
## 500 cases
## 2721 instances of 7 activities
## 7 resources
## Events occurred from 2017-01-02 11:41:53 until 2018-05-05 07:16:02
##
## Variables were mapped as follows:
## Case identifier:     patient
## Activity identifier:     handling
## Resource identifier:     employee
## Activity instance identifier:    handling_id
## Timestamp:           time
## Lifecycle transition:        registration_type
##
## # A tibble: 5,442 x 7
##    handling patient employee handling_id registration_ty~ time
##    <fct>    <chr>   <fct>    <chr>       <fct>            <dttm>
##  1 Registr~ 1       r1       1           start            2017-01-02 11:41:53
##  2 Registr~ 2       r1       2           start            2017-01-02 11:41:53
##  3 Registr~ 3       r1       3           start            2017-01-04 01:34:05
##  4 Registr~ 4       r1       4           start            2017-01-04 01:34:04
##  5 Registr~ 5       r1       5           start            2017-01-04 16:07:47
##  6 Registr~ 6       r1       6           start            2017-01-04 16:07:47
##  7 Registr~ 7       r1       7           start            2017-01-05 04:56:11
##  8 Registr~ 8       r1       8           start            2017-01-05 04:56:11
##  9 Registr~ 9       r1       9           start            2017-01-06 05:58:54
## 10 Registr~ 10      r1       10          start            2017-01-06 05:58:54
## # ... with 5,432 more rows, and 1 more variable: .order <int>
```

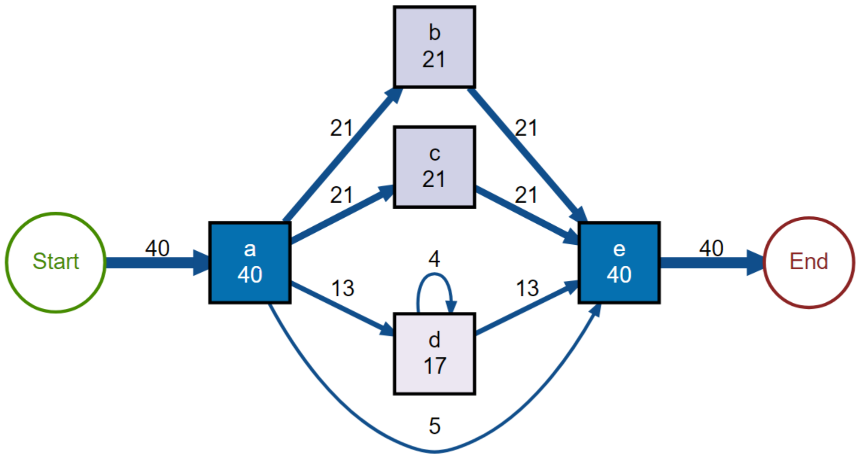
A Causal net can be created as follows.

```r
# Causal graph / Heuristics net
causal_net(patients)
```



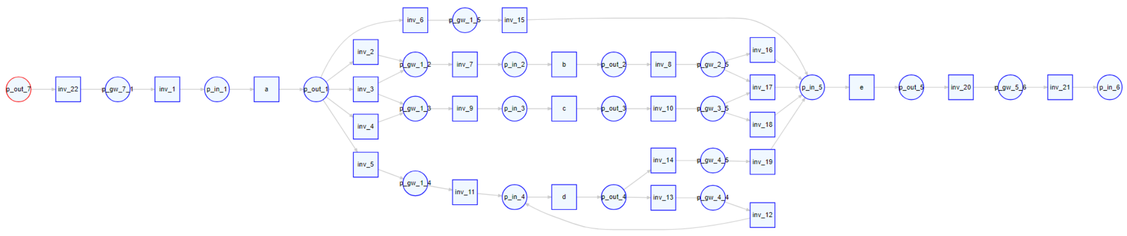
The following is the causal net for the built-in L_heur_1 event log that, was introduced in the **Process Mining book**:

```r
# Example from Process mining book
causal_net(L_heur_1, threshold = .7)
```



The Causal net can be converted to a Petri net (note that there are some unnecessary invisible transition that are not yet removed):

```r
# Convert to Petri net
library(petrinetR)
cn <- causal_net(L_heur_1, threshold = .7)
pn <- as.petrinet(cn)
render_PN(pn)
```



The Petri net can be further used, for example for conformance checking through the pm4py package (Note that the final marking is currently not saved in petrinetR):

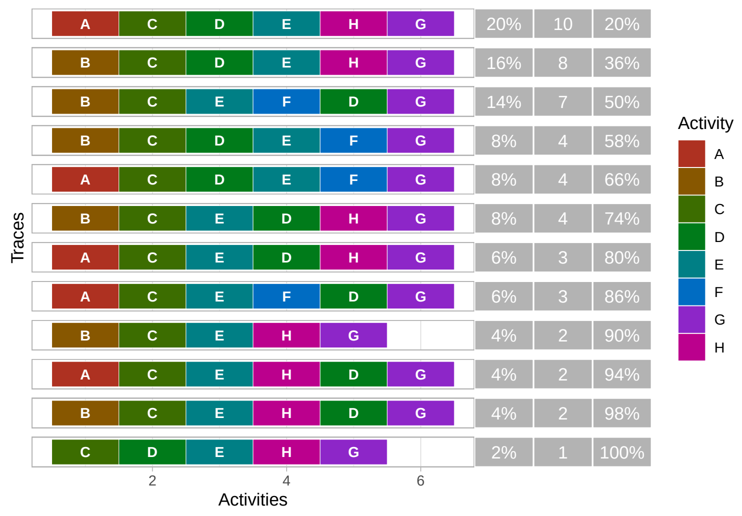
```
library(pm4py)
conformance_alignment(L_heur_1, pn,
                      initial_marking = pn$marking,
                      final_marking = c("p_in_6"))  -> alignments

head(alignments)
```

```
##   case_id log_id model_id log_label model_label cost visited_states
## 1 Case2.0     >>   inv_22        >>        <NA>   10             15
## 2 Case2.0     >>    inv_1        >>        <NA>   10             15
## 3 Case2.0  t_a_0        a         a           a   10             15
## 4 Case2.0     >>    inv_3        >>        <NA>   10             15
## 5 Case2.0     >>    inv_7        >>        <NA>   10             15
## 6 Case2.0     >>    inv_9        >>        <NA>   10             15
##   queued_states traversed_arcs fitness
## 1            44             44       1
## 2            44             44       1
## 3            44             44       1
## 4            44             44       1
## 5            44             44       1
## 6            44             44       1
```

Propro

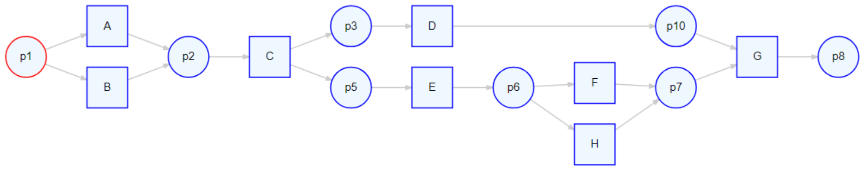
Propro is a package for building probabilistic models on top of process models and event data. In this illustration, we will use the following example log.

```
log <- log_2_paper_ICPM
log %>%
    trace_explorer(coverage = 1)
```



Furthermore, we will use the following model.

```r
net <- model_2_paper_ICPM
net$final_marking <- "p8"
render_PN(net)
```



Constructing a process model starts with the create_propro function, which we can view by printing it.

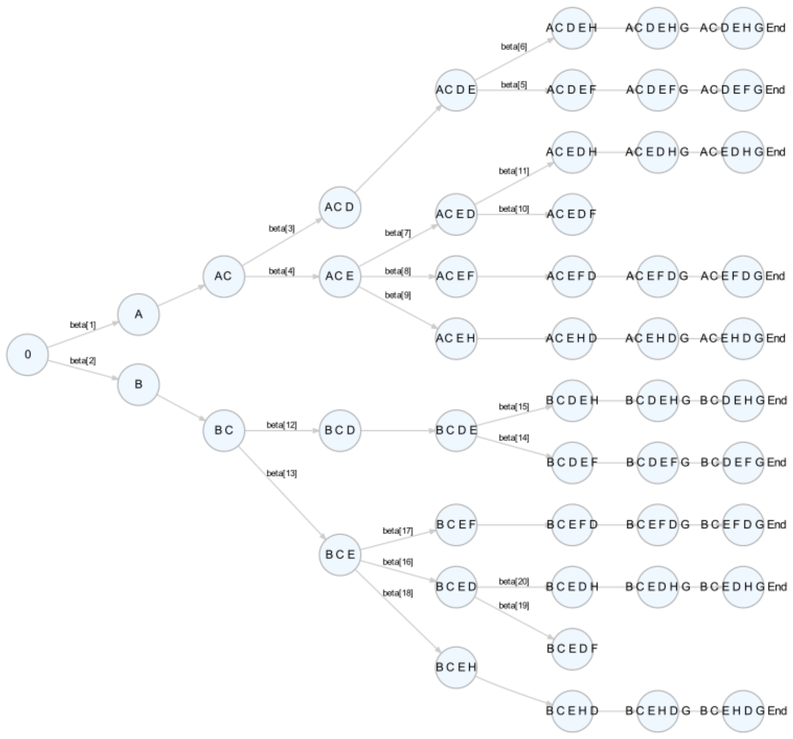
```r
create_propro(log, net) -> propro
```

```r
print_propro(propro)
```

```
## model{
##
## y[1:12] ~ dmulti(theta[1:12], N)
##
## theta[1] <- beta_f*beta[2]*beta[13]*beta[18]
## theta[2] <- beta_f*beta[2]*beta[13]*beta[17]
## theta[3] <- beta_f*beta[2]*beta[13]*beta[16]*beta[20]
## theta[4] <- beta_f*beta[2]*beta[12]*beta[15]
## theta[5] <- beta_f*beta[2]*beta[12]*beta[14]
## theta[6] <- beta_f*beta[1]*beta[4]*beta[9]
## theta[7] <- beta_f*beta[1]*beta[4]*beta[8]
## theta[8] <- beta_f*beta[1]*beta[4]*beta[7]*beta[11]
## theta[9] <- beta_f*beta[1]*beta[3]*beta[6]
## theta[10] <- beta_f*beta[1]*beta[3]*beta[5]
## theta[11] <- (1-beta_f)
## theta[12] <- beta_f*beta[2]*beta[13]*beta[16]*beta[19] + beta[1]*beta[4]*beta[7]*beta[10]
##
## beta[1]
## beta[2]
## beta[3]
## beta[4]
## beta[5]
## beta[6]
## beta[7]
## beta[8]
## beta[9]
## beta[10]
## beta[11]
## beta[12]
## beta[13]
## beta[14]
## beta[15]
## beta[16]
## beta[17]
## beta[18]
## beta[19]
## beta[20]
## beta_f
##
## }
```

In order to see what the different beta's refer to, we can plot the underlying automaton

```
plot_automaton(propro)
```



We now have to specify the priors. Let's start by automatically setting the complements of all splits which have two options.

```
propro %>%
    set_prior_complements(n = 2) -> propro
```

```
## Joining, by = "choice_id"
```

```
list_priors(propro)
```

```
## # A tibble: 21 x 3
##    prior    choice_id specification
##    <chr>        <int> <chr>
##  1 beta[1]          1 <NA>
##  2 beta[2]          1 <- 1 - (beta[1])
##  3 beta[3]          2 <NA>
##  4 beta[4]          2 <- 1 - (beta[3])
##  5 beta[5]          3 <NA>
##  6 beta[6]          3 <- 1 - (beta[5])
##  7 beta[7]          4 <NA>
##  8 beta[8]          4 <NA>
##  9 beta[9]          4 <NA>
## 10 beta[10]         5 <NA>
## # ... with 11 more rows
```

Furthermore, we can see that some probabilities should be the same if we interpret the petri net strictly. We therefore implement the following constraints.

- beta 3 = beta 12
- beta 5 = beta 10 = beta 14 = beta 19
- beta 8 = beta 17
- beta 7 = beta 16
- beta 9 = beta 18

```
propro %>%
    set_prior("beta[12]", "<- beta[3]")%>%
    set_prior("beta[10]", "<- beta[5]")%>%
    set_prior("beta[14]", "<- beta[5]")%>%
    set_prior("beta[19]", "<- beta[5]")%>%
    set_prior("beta[17]", "<- beta[8]")%>%
    set_prior("beta[16]", "<- beta[7]")%>%
    set_prior("beta[18]", "<- beta[9]") -> propro
```

Now we can define the remaining priors. For beta 7,8 and 9, we will use a Dirichlet distributions. Therefore, we first combine these into one prior specification. Then we define the distribution and add alpha to the data.

```
propro %>%
    combine_consecutive_priors(start = 7, end = 9) %>%
    set_prior("beta[7:9]", "~ ddirich(alpha[1:3])") %>%
    add_data("alpha", c(1,1,1)) -> propro
```

```
list_priors(propro)
```

```
## # A tibble: 19 x 4
##    prior     choice_id specification             nr
##    <glue>        <int> <chr>                  <dbl>
##  1 beta[1]           1 <NA>                       1
##  2 beta[2]           1 <- 1 - (beta[1])           2
##  3 beta[3]           2 <NA>                       3
##  4 beta[4]           2 <- 1 - (beta[3])           4
##  5 beta[5]           3 <NA>                       5
##  6 beta[6]           3 <- 1 - (beta[5])           6
##  7 beta[10]          5 <- beta[5]                10
##  8 beta[11]          5 <- 1 - (beta[10])         11
##  9 beta[12]          6 <- beta[3]                12
## 10 beta[13]          6 <- 1 - (beta[12])         13
## 11 beta[14]          7 <- beta[5]                14
## 12 beta[15]           7 <- 1 - (beta[14])        15
## 13 beta[16]           8 <- beta[7]               16
## 14 beta[17]           8 <- beta[8]               17
## 15 beta[18]           8 <- beta[9]               18
## 16 beta[19]           9 <- beta[5]               19
## 17 beta[20]           9 <- 1 - (beta[19])        20
## 18 beta_f            NA <NA>                     NA
## 19 beta[7:9]         NA ~ ddirich(alpha[1:3])    NA
```

All remaining priors we will set to beta distribution with parameters a = 1 and b = 1.

```
propro %>%
    set_prior("beta[1]", "~dbeta(1,1)")%>%
    set_prior("beta[3]", "~dbeta(1,1)")%>%
    set_prior("beta[5]", "~dbeta(1,1)")%>%
    set_prior("beta_f", "~dbeta(1,1)") -> propro
```

Finally, let's add additional variable. For example, a delta which compares beta[5] with beta[8]. Then we save the propro model.

```
propro %>%
    add_variable("delta[1]", "<- beta[6] - beta[9]") -> propro
```

The final prior specification looks as follows.

```
list_priors(propro)
```

```
## # A tibble: 19 x 4
##    prior     choice_id specification             nr
##    <glue>        <int> <chr>                  <dbl>
##  1 beta[1]           1 ~dbeta(1,1)                1
##  2 beta[2]           1 <- 1 - (beta[1])           2
##  3 beta[3]           2 ~dbeta(1,1)                3
##  4 beta[4]           2 <- 1 - (beta[3])           4
##  5 beta[5]           3 ~dbeta(1,1)                5
##  6 beta[6]           3 <- 1 - (beta[5])           6
##  7 beta[10]          5 <- beta[5]                10
##  8 beta[11]          5 <- 1 - (beta[10])         11
##  9 beta[12]          6 <- beta[3]                12
## 10 beta[13]          6 <- 1 - (beta[12])         13
## 11 beta[14]          7 <- beta[5]                14
## 12 beta[15]          7 <- 1 - (beta[14])         15
## 13 beta[16]          8 <- beta[7]                16
## 14 beta[17]          8 <- beta[8]                17
## 15 beta[18]          8 <- beta[9]                18
## 16 beta[19]          9 <- beta[5]                19
## 17 beta[20]          9 <- 1 - (beta[19])         20
## 18 beta_f           NA ~dbeta(1,1)               NA
## 19 beta[7:9]        NA ~ ddirich(alpha[1:3])     NA
```

We can now run the model, after writing it to a file. The resulting posterior distributions can then be used for analysis.

```
propro %>%
    write_propro("propro_model2.txt") %>%
    run_propro(n.chains = 2, n.iter = 40000, n.burnin = 1000) -> results
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 1
##    Unobserved stochastic nodes: 5
##    Total graph size: 34
##
## Initializing model
```

For example, let's look at the posterior distribution of the delta parameter we added.

```
results$BUGSoutput$summary["delta[1]",]
```

```
##          mean           sd         2.5%          25%          50%          75%
##    0.5331691    0.1106419    0.2941453    0.4611763    0.5387333    0.6139176
##        97.5%         Rhat        n.eff
##    0.7268318    1.0011404 2000.0000000
```

Because the value zero does not lie within the 95% confidence interval, we reject the hypothesis that beta[6] and beta[9] are the same. In other words, the probability of observing H is significantly different after ACDE compared to after ACE.

CollaborateR

In order to illustrate the creation of the collaboration graphs, we will use the following example data set of a version control system log. The dataset contains information on commits, related to a specific file, and executed by a specific programmer. A commit can add, modify or delete a file.

```
data("vcs_log")
vcs_log
```

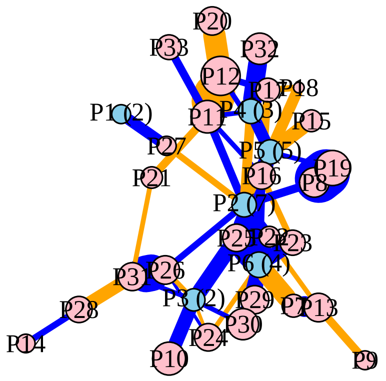
```
## # A tibble: 2,423 x 7
##    commitID fileId programmerID modifierStatus timestamp
##    <chr>     <int>        <dbl> <chr>          <dttm>
##  1 1001        877           46 Modified       2019-02-05 19:38:03
##  2 1001       1764           46 Deleted        2019-02-05 19:38:03
##  3 1001        877           47 Modified       2019-02-05 19:38:03
##  4 1001       1764           47 Deleted        2019-02-05 19:38:03
##  5 1002       1112           22 Modified       2019-03-22 04:49:39
##  6 1002        236           22 Deleted        2019-03-22 04:49:39
##  7 1002        701           22 Added          2019-03-22 04:49:39
##  8 1003        361           32 Added          2019-08-27 13:25:54
##  9 1003        167           32 Added          2019-08-27 13:25:54
## 10 1003        327           32 Modified       2019-08-27 13:25:54
## # ... with 2,413 more rows, and 2 more variables: programmerName <fct>,
## #   event <chr>
```

Using the build_graph function, the collaboration graph can be created from the data. More information on the different settings can be found in the **original paper**

```
g <- build_graph(vcs_log, useFileImportance = TRUE,
                 AgVP = 0.45, AgCP = 0.25,
                 AbVP = 0.1, AbCP = 0.4)
```

The graph can then be visualized as follows.

```
visualize_graph(g,anonymize = TRUE)
```



ProcessanimateR

Examples of processanimateR — as they are not ideal to be shown in PDF format — can be found **here**

More information

For more information and examples of the bupaR ecosystem, we refer to the website **www.bupar.net**