Technische Universiteit
Eindhoven
University of Technology

Den Dolech 2, 5612 AZ Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

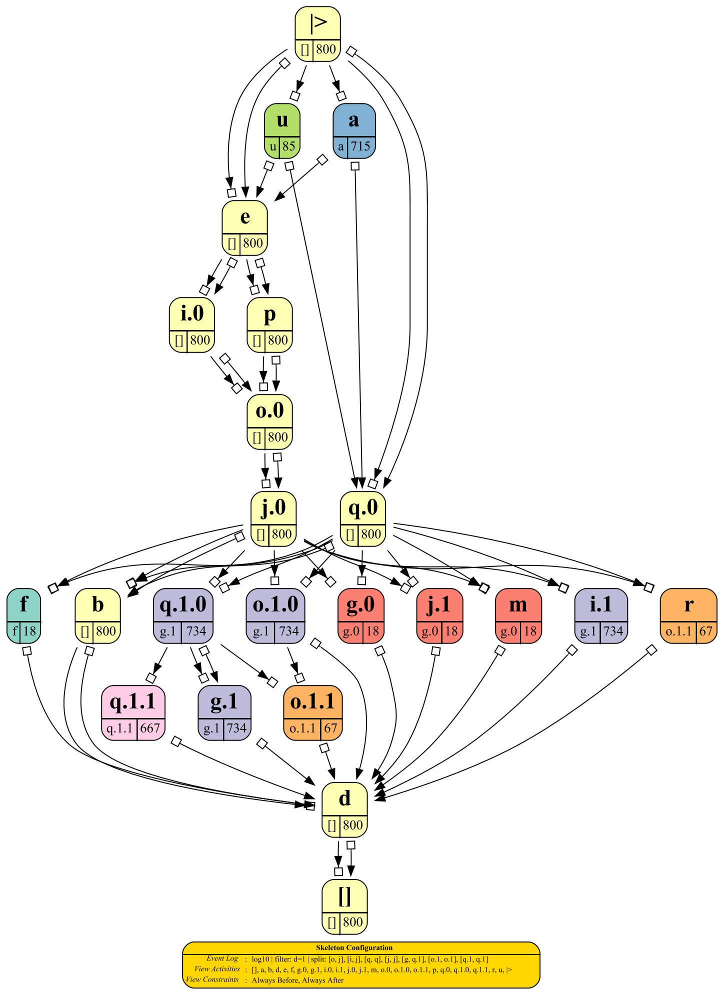
Author
Eric Verbeek

Date
July 14, 2017

Version
1.1

Log Skeletons

Contribution to the Process Discovery Contest 2017



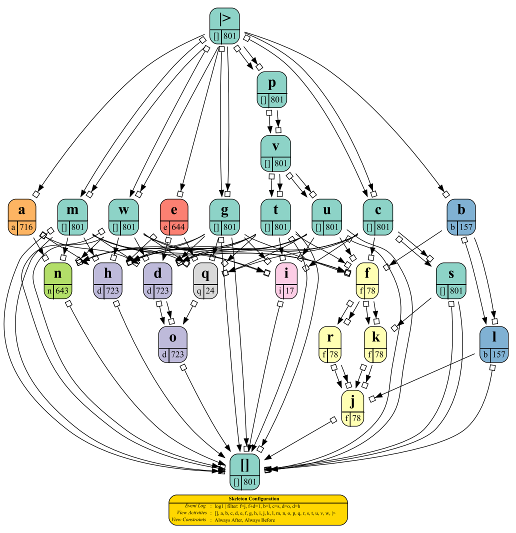
Skeleton Configuration

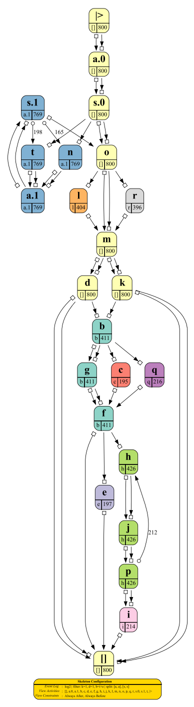
Event Log : log10 | filter: d=1 | split: [o, j], [i, j], [q, q], [j, j], [g, q.1], [o.1, o.1], [q.1, q.1]
View Activities : [], a, b, d, e, f, g.0, g.1, i.0, i.1, j.0, j.1, m, o.0, o.1.0, o.1.1, p, q.0, q.1.0, q.1.1, r, u, |>
View Constraints : Always Before, Always After

1 Introduction

This document explain my contribution to the Process Discovery Contest 2017. Last year, I competed using a decomposition approach, this year I thought to do it radically different. Many of the discovery algorithm focus heavily on the directly-follows relation, and for many of them this is a black-and-white relation: Either a is directly followed by b, or not. This way, these discovery algorithm abstract from information that could potentially be very useful for discovery, and for classification.

For this reason, I've tried to think out-of-the-box, by using frequencies of activities, and by using different relations than only the directly-follows relation. The resulting model I've called a *log skeleton*, as it more or less shows the structure of the log in a comprehensive way (IMO). The figure below shows two views on such log skeletons: The one on the left-hand side for $log1$ and the other on the right-hand side for $log2$. Later on, the meaning of the arcs will be explained in details, for now it suffices to know that they indicate a necessary relation between two activities.



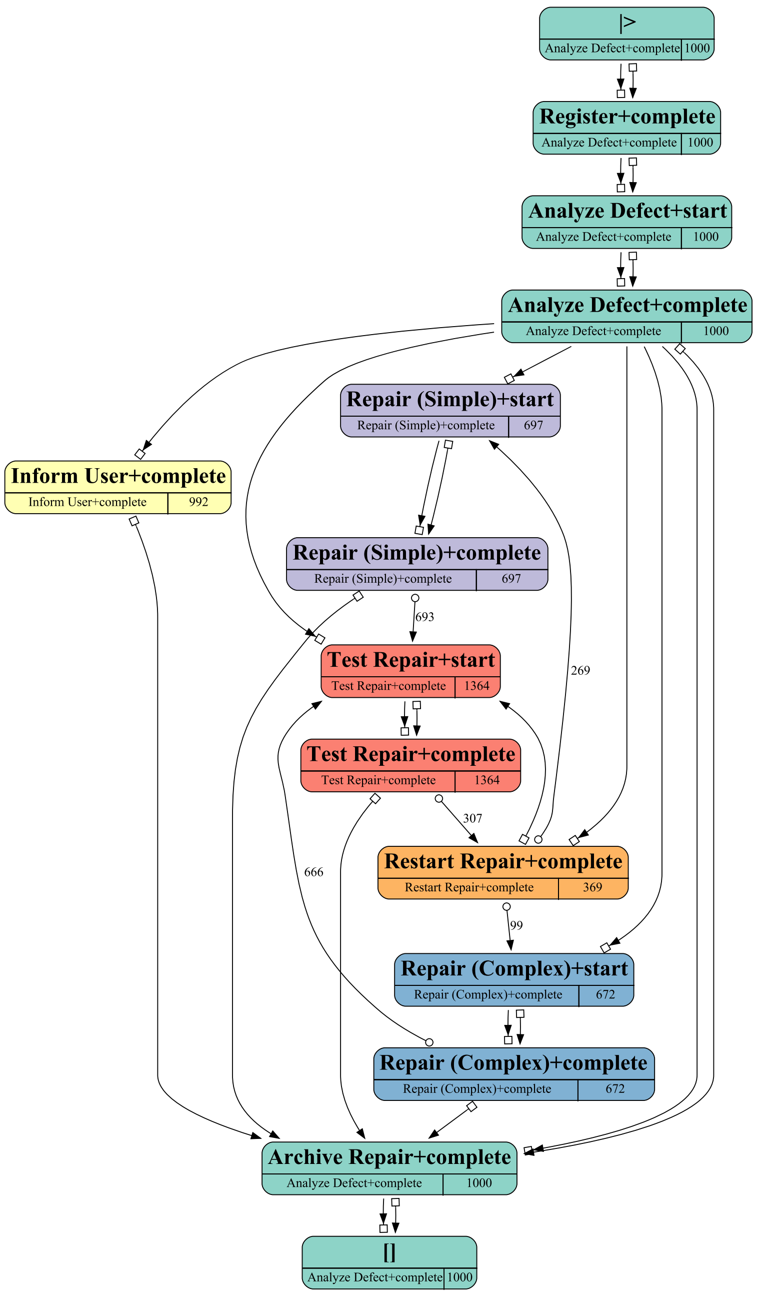


Sometimes, the conversion from a log skeleton to a Petri net is quite simple, of which $log2$ is a nice example. By unfolding its initial loop once, and by looking at the frequencies of the activities, this conversion can be quite easily done. However, for some other logs this conversion much less straightforward, of which $log1$ is a nice example. Especially the arrows (long-term dependencies?) from a to n and the arcs going into the yellow $f - k - r - j$ block are hard to do. It is possible, but the Petri net model does not look as nice (again, IMO) as the log skeleton does.

Using only the log skeletons, I've constructed a classifier for the test logs. Nevertheless, it must be mentioned that I'm using quite a lot of log skeletons for the classification as I allow the classifier two take two arbitrary activities from the event log and to filter the event log on these activities, where each activity may be required (only traces where this activity occurs are filtered in) or forbidden (only traces that where this activity does not occur are filtered in). For each of these filtered logs, I then construct a log skeleton that I use in the classification. As a result, it is clear that I need the original event log for the classification, as the log skeleton of this event log does not suffice for the filtering. As such, one could argue that I'm actually using the original event log as my model. Fair enough. But it classifies quite good, and although it is called a Process Discovery Context, it is more a Process Classification Contest.

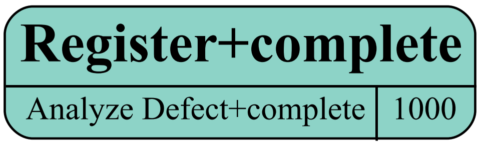
The remainder of this report is organized as follows. Chapter 2 introduces the log skeletons and their replay semantics, if one would like to call it that. Chapter 3 details the process of discovering a nice log skeleton from an event log by means of an example: *log10*, and shows the nice log skeletons we obtained this way for every log. This Chapter shows that in the process we will massage a log into a more favorable shape (by removing noisy traces and applying so-called splitters). Chapter 4 details the approach taken for the classification, which, as mentioned, uses lots of log skeletons. But as these log skeletons can be created very fast, this approach is very well feasible. Chapter 5 details the implementation of the entire approach using log skeletons, which is done in a new ProM 6 package. Chapter 6 details the results obtained using the implemented approach. Finally, Chapter 7 ends the report with some conclusions and observations.

2 Log Skeleton


|>
Analyze Defect+complete 1000
Register+complete
Analyze Defect+complete 1000
Analyze Defect+start
Analyze Defect+complete 1000
Analyze Defect+complete
Analyze Defect+complete 1000
Repair (Simple)+start
Repair (Simple)+complete 697
Inform User+complete
Inform User+complete 992
Repair (Simple)+complete
Repair (Simple)+complete 697
693
Test Repair+start
Test Repair+complete 1364
269
Test Repair+complete
Test Repair+complete 1364
307
666
Restart Repair+complete
Restart Repair+complete 369
99
Repair (Complex)+start
Repair (Complex)+complete 672
Repair (Complex)+complete
Repair (Complex)+complete 672
Archive Repair+complete
Analyze Defect+complete 1000
[]
Analyze Defect+complete 1000

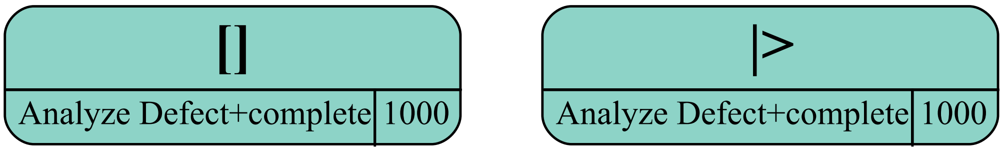
This shows a typical log skeleton. A log skeleton is a graph, where every node corresponds to an activity and every edge corresponds to a constraint between two (not necessarily different) nodes.

2.1 Activities



This shows a typical node, which corresponds to an activity. In this case, the activity is $\mathit{Register} + \mathit{complete}$, which has occurred 1000 times in the log, and has $\mathit{AnalyzeDefect} + \mathit{complete}$ as equivalence class. This equivalence class will be explained later on.

Apart from the regular activities, a log skeleton contains two artificial activities: $| >$ (start of a trace) and $[]$ (end of a trace).



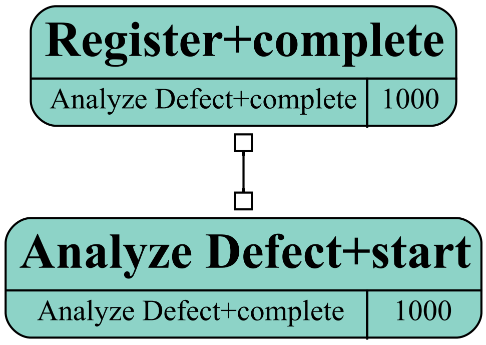
This shows that the log contains 1000 traces.

2.2 Constraints

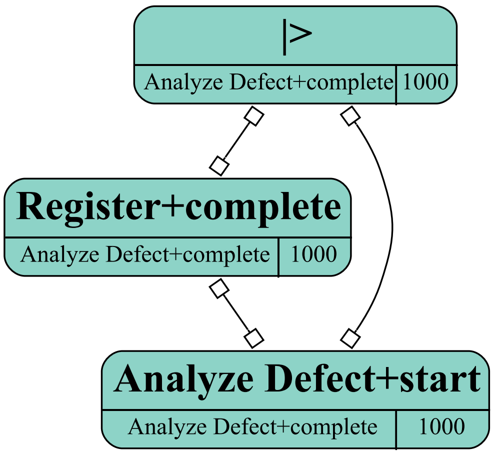
A log skeleton may contain six different edge types: One edge type for every possible constraint. Possible constraints are $\mathit{Always\ Together}$, $\mathit{Always\ Before}$, $\mathit{Always\ After}$, $\mathit{Never\ Together}$, Next $(\mathit{One\ Way})$, and Next $(\mathit{Both\ Ways})$.

2.2.1 Always Together

The $\mathit{Always\ Together}$ constraint is visualized by an open box on each end of the edge. In a log skeleton, an open box roughly translates to Always, and the end at which it is placed determines the viewpoint for the constraint. As this constraint is symmetrical, it is placed at both edge ends.



This shows an $\mathit{Always\ Together}$ constraint between $\mathit{Register} + \mathit{complete}$ and $\mathit{AnalyzeDefect} + \mathit{start}$. This constraint indicates that both activities occur equally often in every trace. As a result, if $\mathit{Register} + \mathit{complete}$ occurs n times in some trace, then so does $\mathit{AnalyzeDefect} + \mathit{start}$.

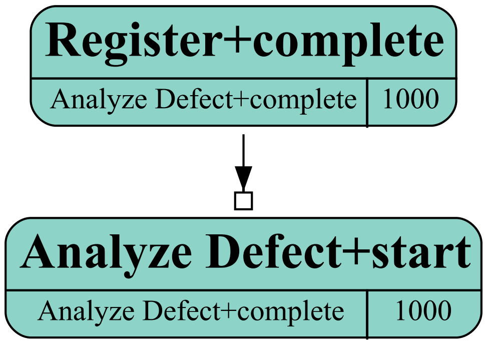

|>
Analyze Defect+complete 1000
Register+complete
Analyze Defect+complete 1000
Analyze Defect+start
Analyze Defect+complete 1000

This shows that both activities are *Always Together* with the artificial start activity $| >$. From this, we may conclude that both activities occur exactly once in every trace.

This constraint determines the equivalence class of an activity: Two activities related by the *Always Together* constraint are considered to be equivalent. As a result, they share the same color in the visualization of the log skeleton.

2.2.2 Always Before

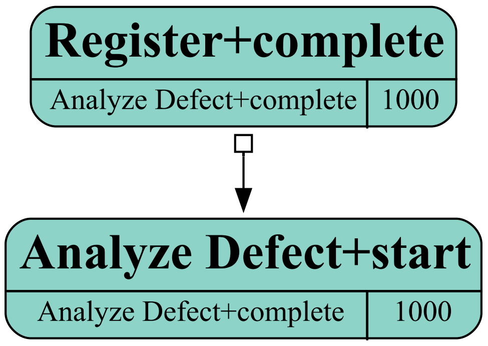
The *Always Before* constraint is visualized by an arrow with the open box (*Always*) at the head.



This indicates that every occurrence of $AnalyzeDefect + start$ is preceded in the trace by an occurrence of $Register + complete$: If you stand on an $AnalyzeDefect + start$, and look towards the start of that trace, you will see a $Register + complete$ somewhere.

2.2.3 Always After

The *Always After* constraint is visualized by an arrow with the open box (*Always*) at the tail.

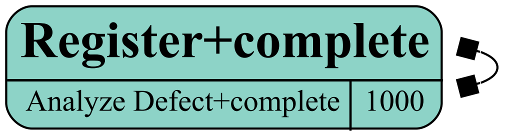


This indicates that every occurrence of $Register + complete$ is followed in the trace by an occurrence of $AnalyzeDefect + start$: If you stand on a $Register + complete$, and look towards the end of that trace, you will see an $AnalyzeDefect + start$ somewhere.

Both the *Always Before* and the *Always After* constraints provide a sense of direction in the log skeleton. In the example, it is clear that always first $register + complete$ has to occur, after which $AnalyzeDefect + start$ has to occur.

2.2.4 Never Together

The *Never Together* constraint is visualized by a closed box (*Never*) on each end of the edge.

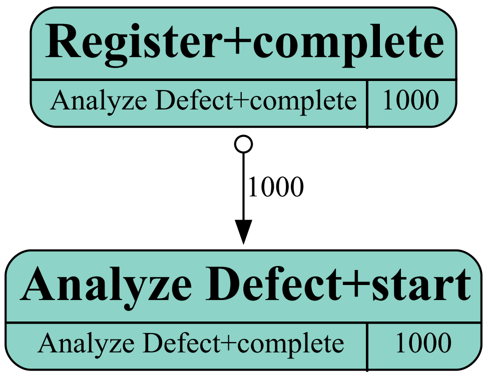


This indicates that $Register + complete$ never occurs together with itself: If you stand on a $Register + complete$, and look at both the start and end of the trace, you will see no $Register + complete$ (note that we assume you cannot see the node you're standing on). As a result, we may conclude that $Register + complete$ occurs at most once in every trace.

The two remaining constraints correspond to the directly-follows-graph, which is well-known in the area of process mining. However, as we think the one-way edges in this graph are more informative than the two-way edges, we have split the edges over the two remaining types.

2.2.5 Next (One Way)

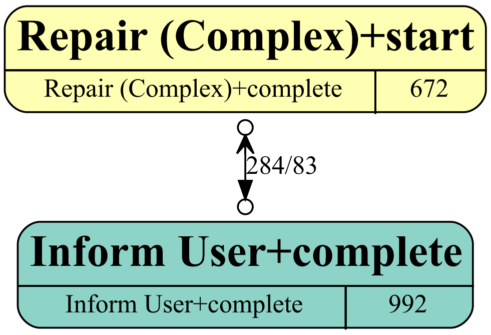
The *Next* (*One Way*) constraint is visualized by an open dot (*Next*) on the tail of the edge.



This indicates that in the log *Register + complete* was 1000 times directly followed by *AnalyzeDefect + start*, and never the other way around.

2.2.6 Next (Both Ways)

The *Next (Both Ways)* constraint is visualized by an open dot (*Next*) on each end of the edge. Furthermore, the arrow at the tail is different to be able to distinguish the source from the target.



This shows that in the log *Repair*(*Complex*) + *start* was 284 times directly followed by *InformUser* + *complete*, and 83 times the other way around.

2.3 Replay Semantics

A trace is accepted by the log skeleton if and only if it does not violate any of the constraints:

Always Together A trace violates an *Always Together* constraint if the activities involved in the constraint do not occur equally often in the trace.

Always Before A trace violates an *Always Before* constraint if some occurrence of the target activity is not preceded by some occurrence of the source activity in the trace.

Always After A trace violates an *Always After* constraint if some occurrence of the source activity is not followed by some occurrence of the target activity in the trace.

Never Together A trace violates a *Never Together* constraint if both activities involved in the constraint occur in the the trace.

Next (One Way) and Next (Both Ways) A trace violates the *Next* constraints if in the trace some activity is directly followed by another constraint, whereas a corresponding *Next* constraint does not exist.

However, we know from the organizers that in a test log precisely 10 traces are non-fitting. Furthermore, we have observed that some of the training logs are not complete in the directly-follows relation: Although in the model some activity could directly be followed by some other activity, the log does not reflect this (in no trace is the first activity directly followed by the second). For this reason, we impose a hierarchy on these constraints, and only check the higher constraints in this hierarchy until at least 10 non-fitting traces were found.

In this constraint hierarchy, we have three different levels, where level 1 is the highest level and level 3 the lowest:

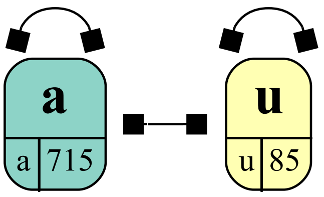
1. *Always Together*

2. *Always Before* and *Always After*
3. *Next* (*One Way*) and *Next* (*Both Ways*)

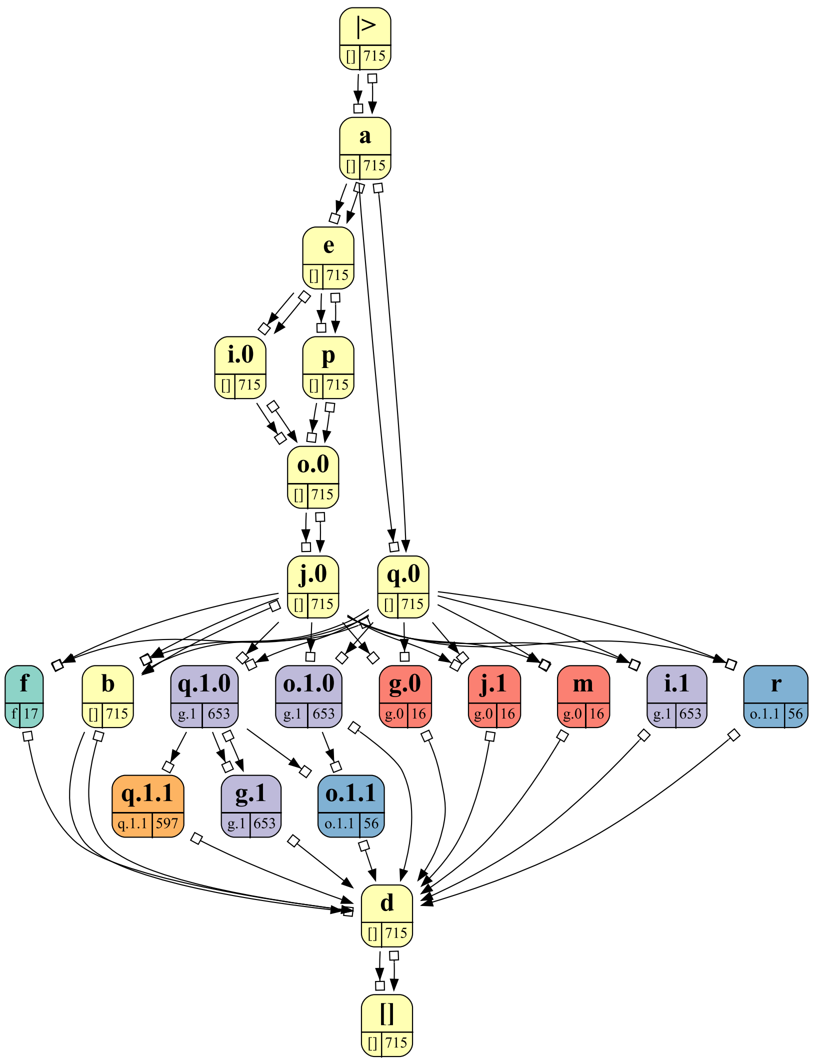
As a result, if we have already found 10 violations to the *Always Together* constraint, we will not check the other constraints for violations.

Note that the *Never Together* constraint is not checked explicitly. To explain this, consider the log skeleton on the front page, and observe the a and u activities that follow the artificial start activity. For these two activities, three *Never Together* constraints are found:

- between a and u,
- between a and itself, and
- between u and itself.



If we combine these three constraints by the fact that there are 800 traces in which a occurs 715 times and u 85 times, we (as humans) immediately 'see' that there is a mandatory choice between a and u. However, to conclude this programmatically, we would need to include a reasoner of sorts. For this reason, we will not check these *Never Together* constraints explicitly, but we will do that in an implicit way by filtering activities. In the example with the a and u activities: If we filter out all traces where u occurs (both in the training log as in the test log), a becomes a mandatory activity:

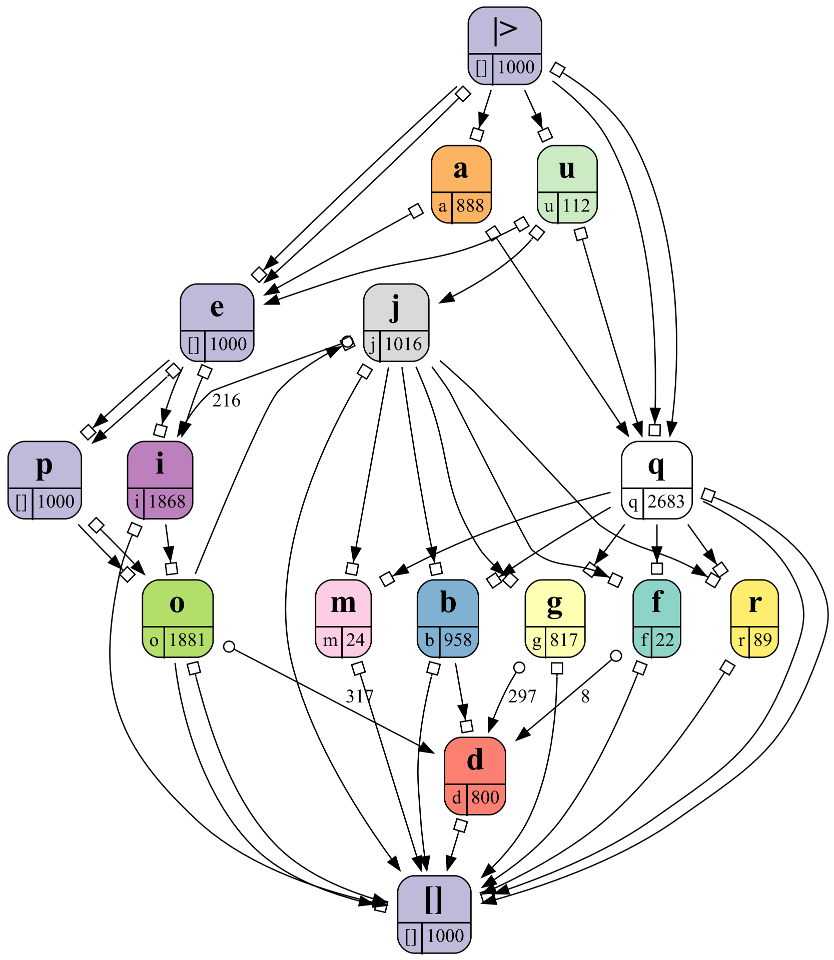


As a result, it will belong to the same equivalence class as the artificial start and end activities, which can then be checked easily by the *Always Together* constraint.

3 Discovery

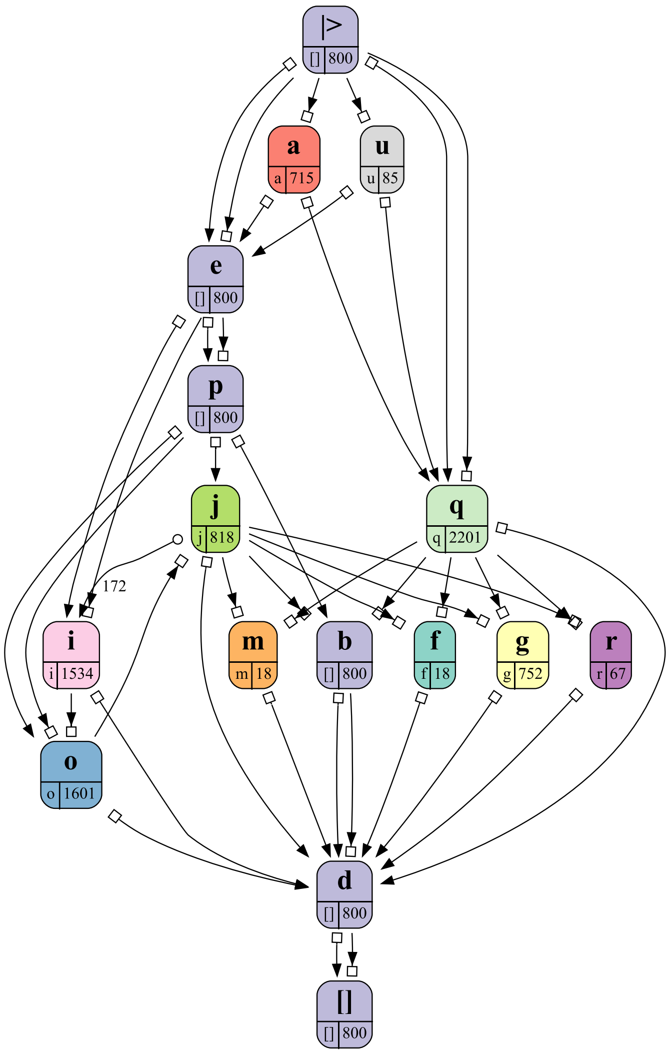
3.1 Example Discovery

To showcase the discovery using log skeletons, we will use it on the *log10* log from the contest.



This shows immediately that e and p occur exactly once in every trace, and that e comes before p. This also hints (as $112+888=1000$) that there is a choice construct right in the beginning between a and u. This can quite easily be confirmed by having a look at the *Never Together* constraint.

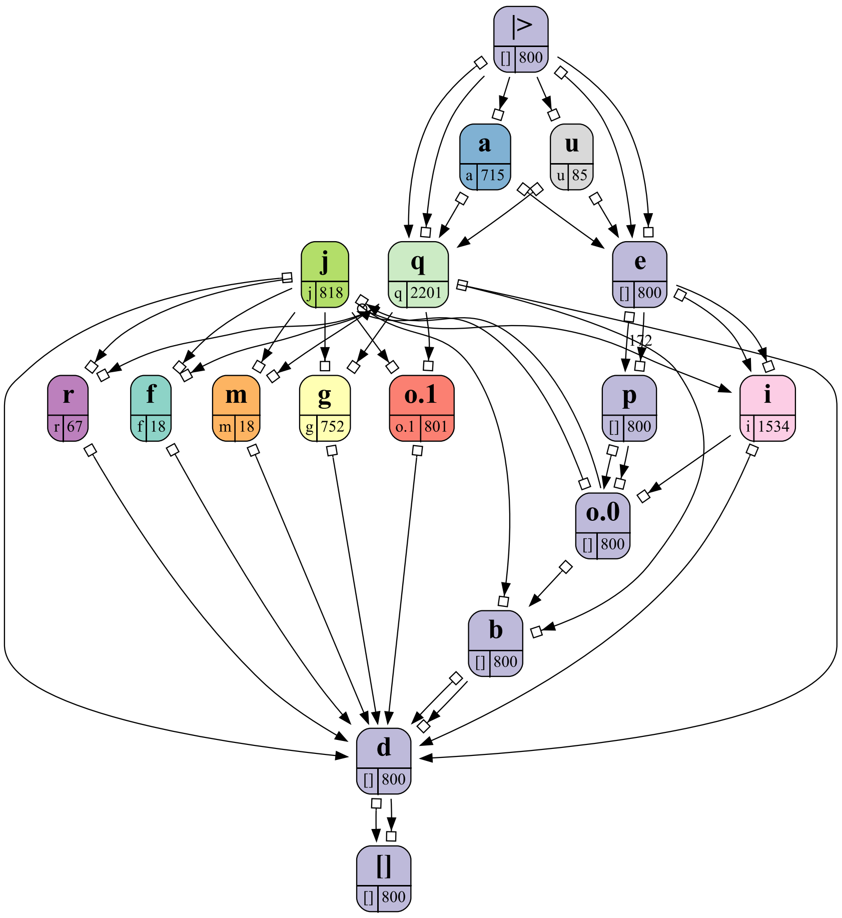
Another thing that is striking is the fact that d, which occurs quite far below in the skeleton, occurs 800 times. From the information as provided by the organizers, we know that *log10* contains noise, which means that the tail of 20% of its traces was removed. That is, from 200 out of 1000 traces, the last part of the trace was removed. Apparently, every time a last part was removed, it included the removal of d. Assuming that this is indeed the case, we filter out those traces that do not contain d, which result in the following log skeleton.



This shows that both b and d now have fallen 'in line' with e, p, $|>$, and $[]$. Note that the *Always Before* constraint between b and p seems to be missing, but observe that this constraint is there by transitivity through j and o.

From the information as provided by the organizers, we also know that the model underlying the log contained reoccurring activities (or duplicate activities). Therefore, we suspect at least i, o, q, and j to be reoccurring. To make the log skeleton more distinctive, and better suited for the classification task that lays ahead, we try to split every such activity into non-reoccurring activities. For this reason, the log skeleton browser contains a simple activity splitter than can easily be configured by the user.

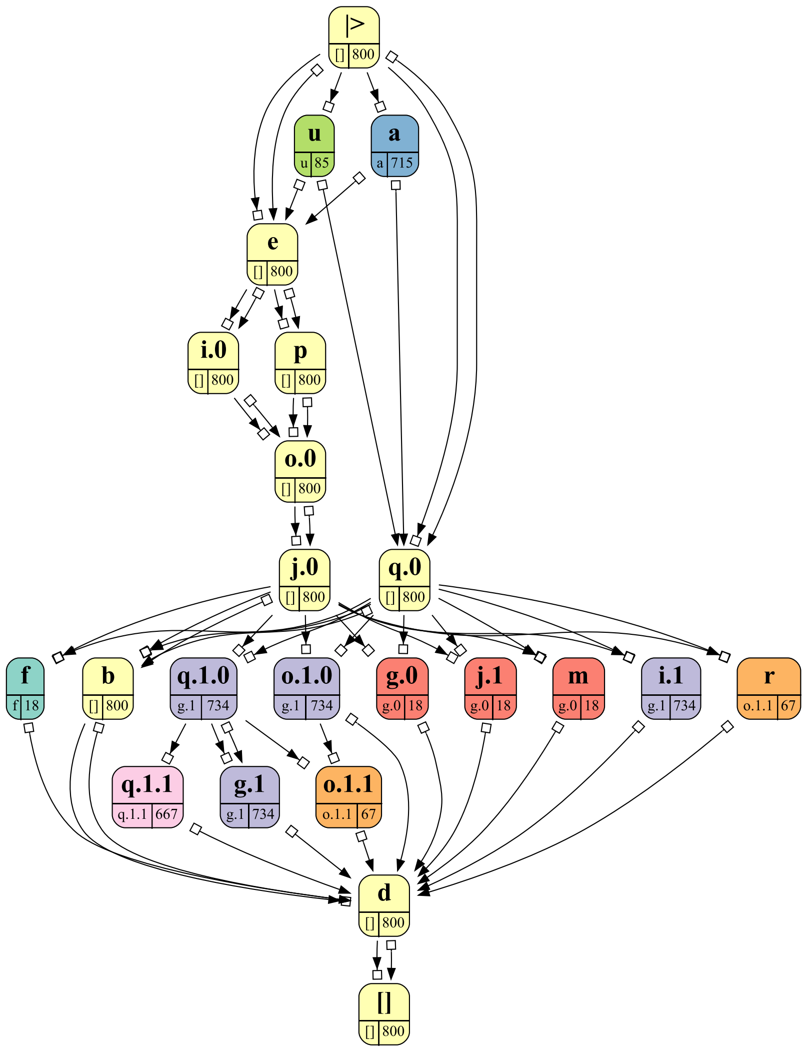
This activity splitter splits some activity, say o, over some other activity, say j. This means that a trace is conceptually split into two parts, where the first part ends with the first occurrence of j, and the second part is the remainder. The activity o is then renamed $o.0$ in the first part of the trace, and renamed $o.1$ in the second part.



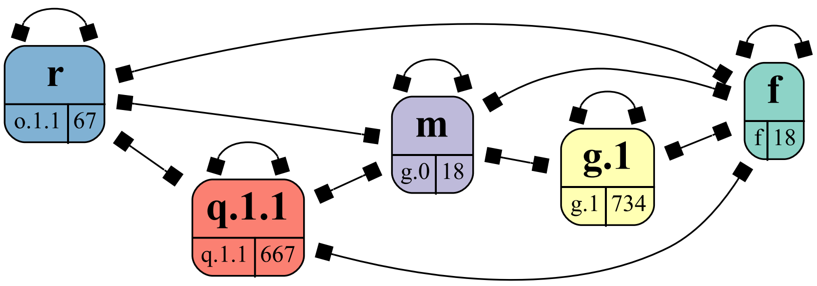
This shows the result of this split. Note that $o.0$ has also fallen ‘in line’, and that now j nicely separates $o.0$ from $o.1$.

The choice to split o over j seems quite arbitrarily chosen, we concede to that, but it is quickly done and a visual inspection of the result may tell whether the split made sense or not. In a way, there is some trial-and-error here, where the user tries out some ways to split activities. Perhaps there are ways to do this in a more automated way, but this has not yet been investigated. At least for this contest, all splits were found using the trail-and-error approach.

As a result of this trial-and-error, we included splitting i over j as well, splitting q over itself (also possible), splitting j over itself, splitting g over $q.1$, splitting $o.1$ over itself, and splitting $q.1$ over itself, which resulted in the following log skeleton.



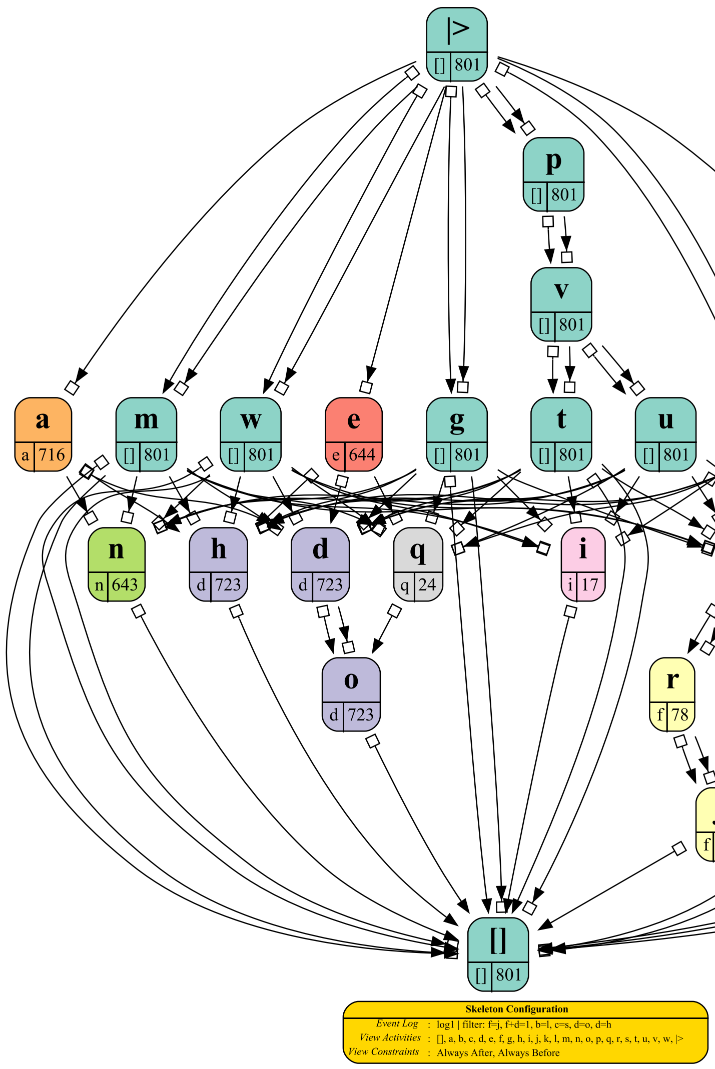
With the help of the other views on the log skeleton, it is possible to extract a Petri net from the log skeleton. For example, the following *Never Together* view on selected activities shows that in the bottom part there is a choice between (1) f, (2) $g.0$, $j.1$, and m, and (3) $g.1$, $i.1$, $o.1.?$, and $q.1.?$, where in the latter cluster there is a choice between either $q.1.1$ or both $o.1.1$ and r.



3.2 Discovered Models

For sake of completeness, every log skeleton in this section comes with its configuration (the golden box below the log skeleton), which shows how the log skeleton was obtained from the original log. Note that we only show the *Always* constraints here (with relevant *Next* (*One Way*) constraints), as these are most informative, but that the other constraints can also be of use.

3.2.1 log1


|>
[] 801
p
[] 801
v
[] 801
a
a 716
m
[] 801
w
[] 801
e
e 644
g
[] 801
t
[] 801
u
[] 801
c
[] 801
b
b 157
n
n 643
h
d 723
d
d 723
q
q 24
i
i 17
f
f 78
s
[] 801
o
d 723
r
f 78
k
f 78
l
b 157
j
f 78
[]
[] 801
Skeleton Configuration
Event Log : log1 | filter: f=j, f+d=1, b=l, c=s, d=o, d=h
View Activities : [], a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, |>
View Constraints : Always After, Always Before

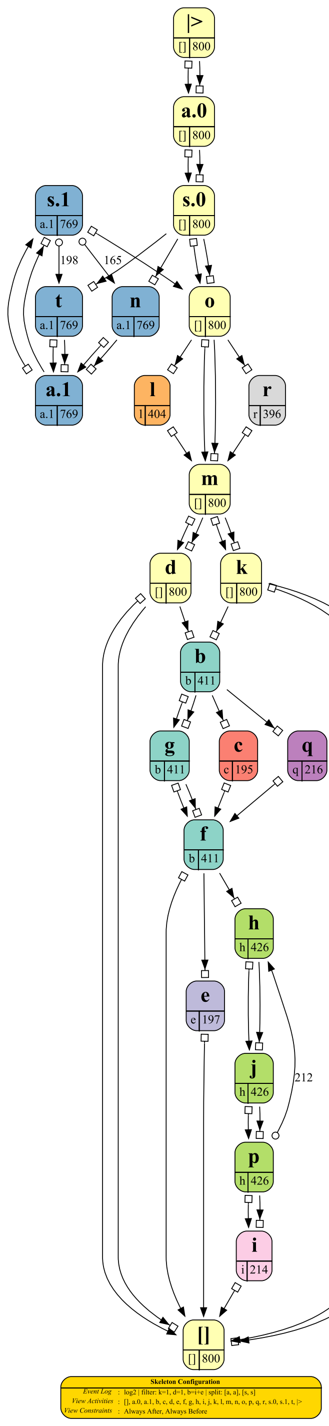
Note the *Always* constraints between a and n, b and c to f, s to k, and l to j. From the information as provided by the organizers, we know that the model underlying this log contains long-term dependencies. We guess these constraints pinpoint some of these long-term dependencies.

Added Test Traces

In the classification process, the trace 4 of the second test event log is added to the training event log. This trace was originally classified as negative, while we assumed it should be positive. By adding it to the training event log, we make sure it will be classified positive. Of course, our assumption could be wrong, in which case the classification will not be better.

Whereas in every trace in the training log every r is always preceded by some l, trace 4 shows that the only r is followed by the only l. As the organizers have confirmed that this trace is indeed a true positive (as we classified all traces correctly for this log, which included a positive classification for this trace), we have the classifier insert the trace into the training log to ensure it will be classified positive.

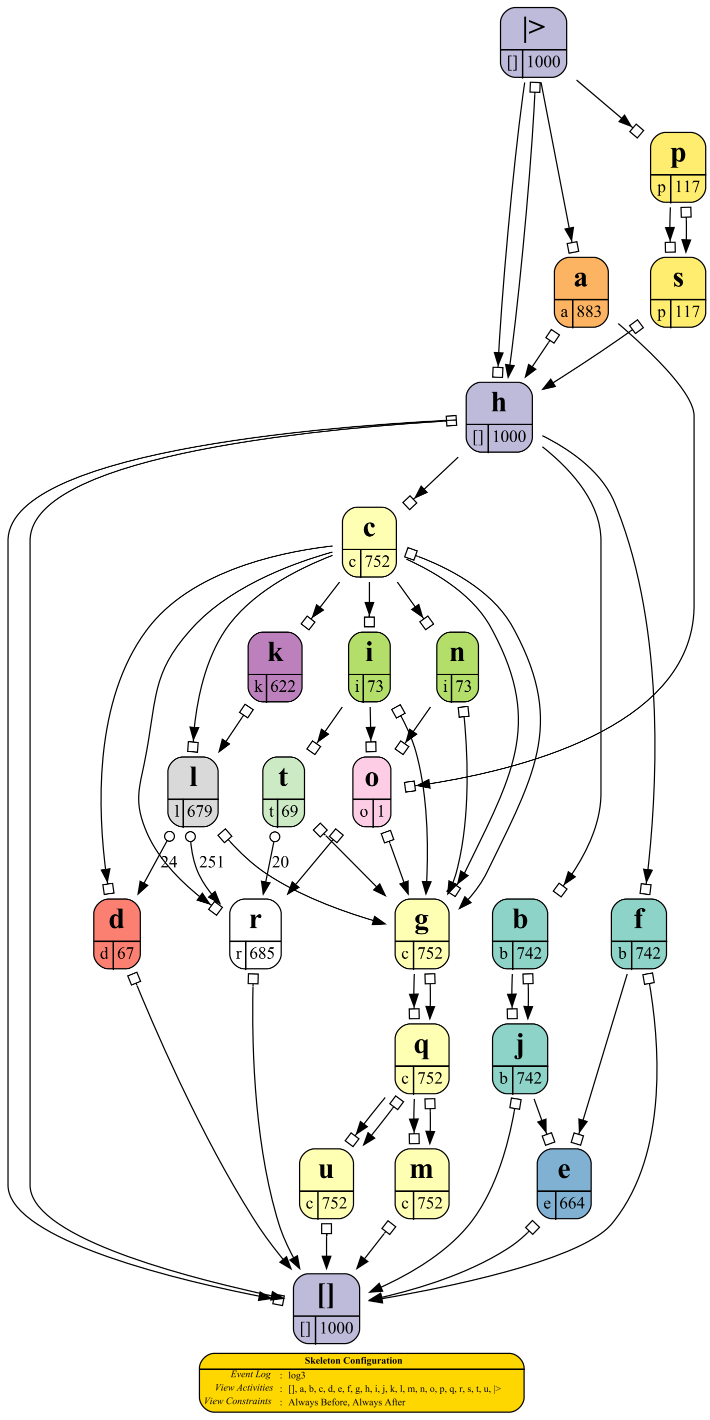
3.2.2 log2



Skeleton Configuration

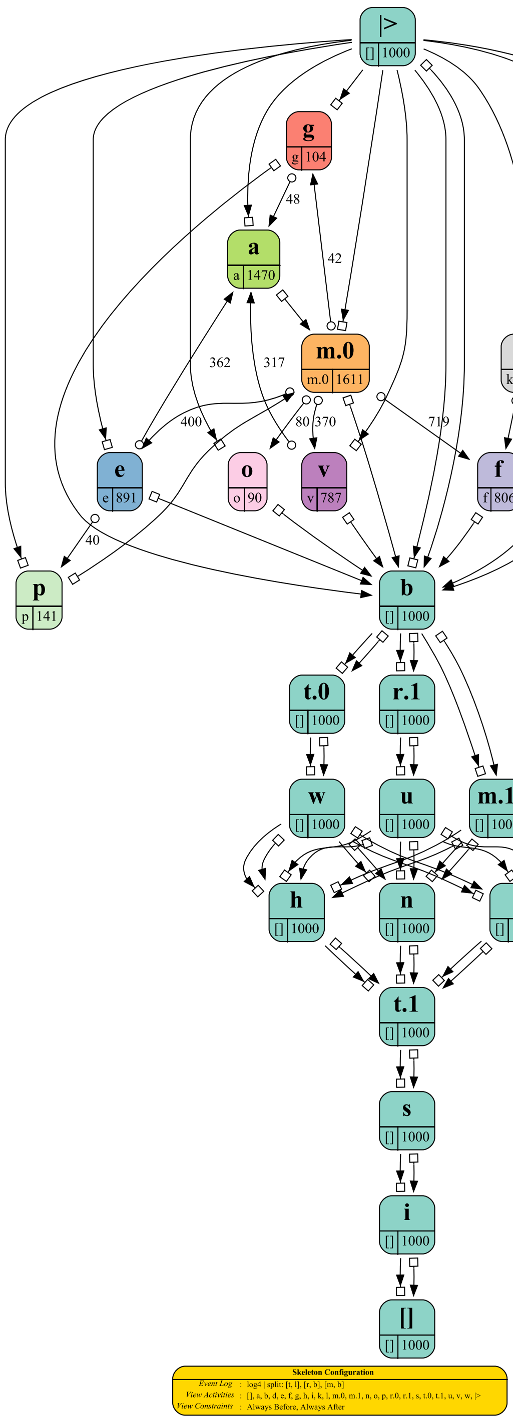
Event Log : log2 | filter: k=1, d=1, b=i+e | split: [a, a], [s, s]
View Activities : [], a.0, a.1, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s.0, s.1, t, |>
View Constraints : Always After, Always Before

3.2.3 log3



Note the fact that o occurs only once in the entire log. Not much conclusions one can draw from that single occurrence.

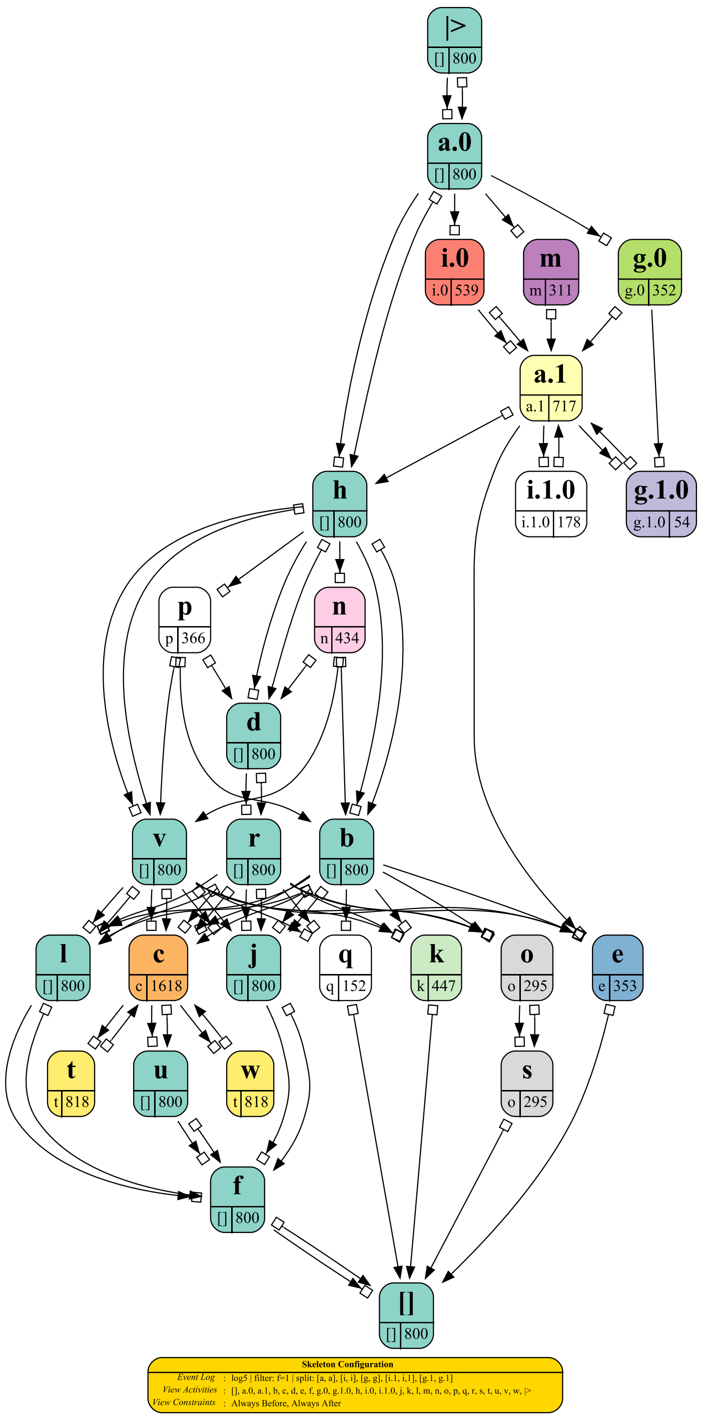
3.2.4 log4



Skeleton Configuration

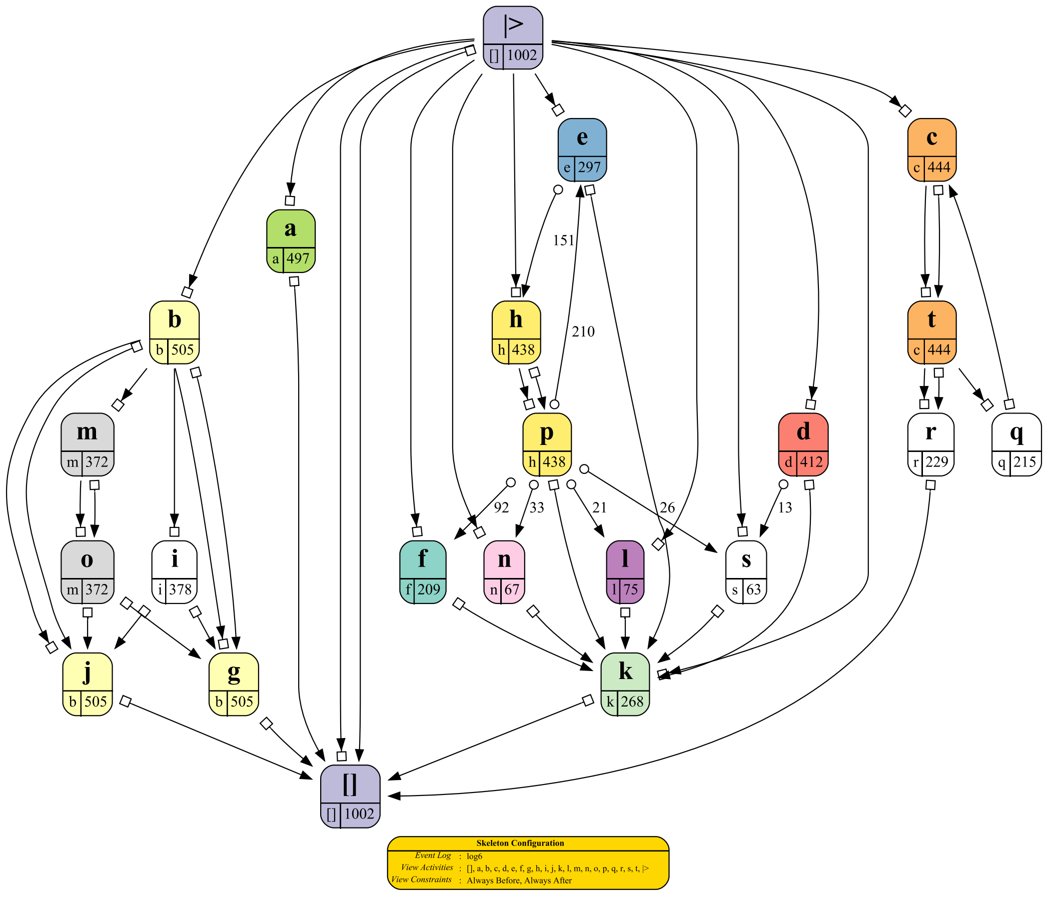
Event Log	:	log4 | split: [t, l], [r, b], [m, b]
View Activities	:	[], a, b, d, e, f, g, h, i, k, l, m.0, m.1, n, o, p, r.0, r.1, s, t.0, t.1, u, v, w, |>
View Constraints	:	Always Before, Always After

3.2.5 log5


|>
[] 800
a.0
[] 800
i.0
i.0 539
m
m 311
g.0
g.0 352
a.1
a.1 717
i.1.0
i.1.0 178
g.1.0
g.1.0 54
h
[] 800
p
p 366
n
n 434
d
[] 800
v
[] 800
r
[] 800
b
[] 800
l
[] 800
c
c 1618
j
[] 800
q
q 152
k
k 447
o
o 295
e
e 353
t
t 818
u
[] 800
w
t 818
s
o 295
f
[] 800
[]
[] 800
Skeleton Configuration
Event Log : log5 | filter: f=1 | split: [a, a], [i, i], [g, g], [i.1, i,1], [g.1, g.1]
View Activities : [], a.0, a.1, b, c, d, e, f, g.0, g.1.0, h, i.0, i.1.0, j, k, l, m, n, o, p, q, r, s, t, u, v, w, |>
View Constraints : Always Before, Always After

An odd construct: *a.1* is in a loop construct, and *e* is always preceded by it, that is, by the loop. Apparently, *e* can only occur if the loop has occurred.

3.2.6 log6

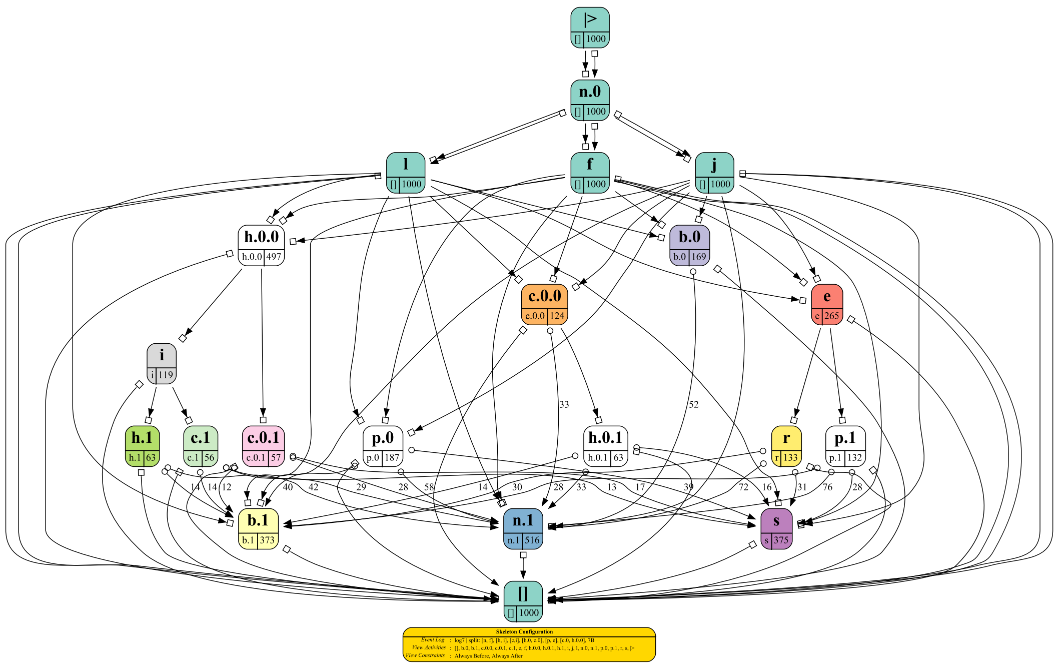


Added Test Traces

In the classification process, the traces 4 and 11 of the second test event log are added to the training event log. These traces were originally classified as negative, while we assumed they should be positive. By adding it to the training event log, we make sure they will be classified positive. Of course, our assumption could be wrong, in which case the classification will not be better.

Whereas in every trace in the training log that contains q, every a is always followed by some c, trace 4 shows that the only a is preceded by both occurrences of c. Whereas in every trace in the training log, every n is always preceded by some a, trace 11 shows that the only n is followed by the only$a$.

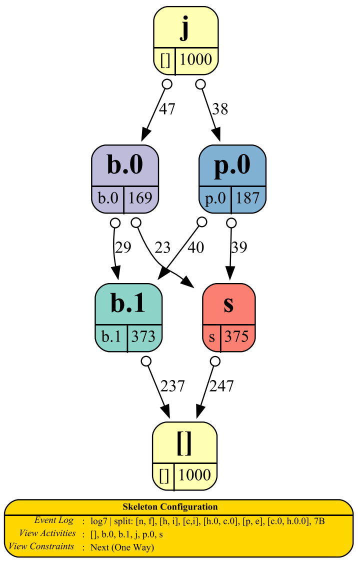
3.2.7 log7



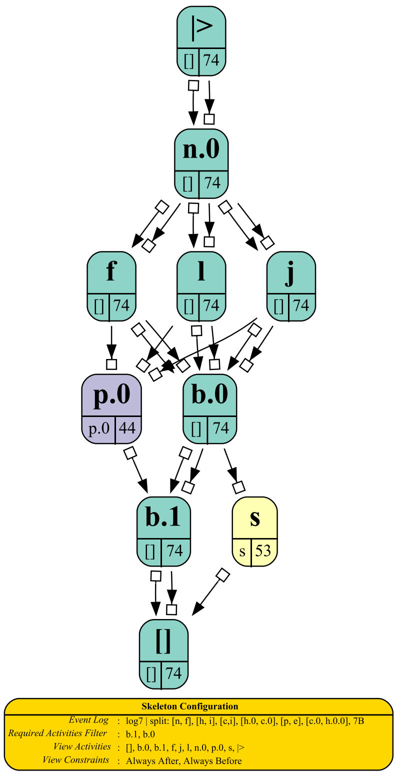
Note that for *log7* we used a splitter named *7B*. This is a splitter which splits activity b using the following rules:

- If the b is the last in the trace, then this b is renamed $b.1$.
- If the b is the last-but-one in the trace while the last one is s, then this b is renamed $b.1$.
- Otherwise, b is renamed $b.0$.

The reason for this less-simple splitter is that I believe the net to end with either an n or an implicit choice between b and s, which many be directly preceded by an implicit choice between b and p. To get some clarity here, we need to split this b into $b.0$ and $b.1$. Clearly, if the b is the last one in the trace, it should be the b in the implicit choice with s. Furthermore, if the next ends with s, the it is OK to consider a last-but-one b to be in the same implicit choice as this s: If we only see p followed by b followed by s, then there is no way to know whether the b is in the same implicit choice with the p or with the s. In all other cases, we may assume that a b should be in the same implicit choice as the p, as the ‘other’ b can only come last or last-but-one.



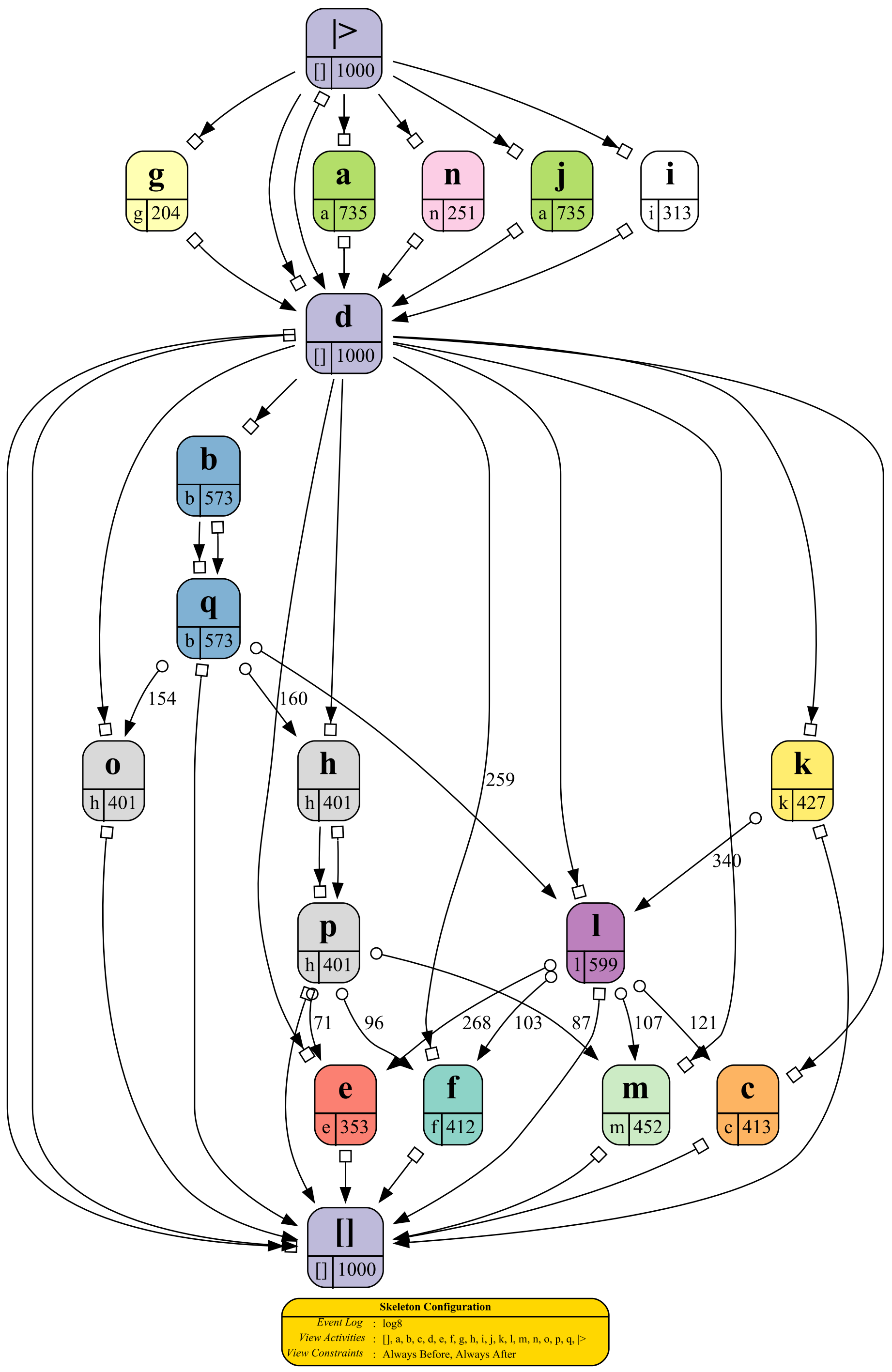
Skeleton Configuration
Event Log : log7 | split: [n, f], [h, i], [c,i], [h.0, c.0], [p, e], [c.0, h.0.0], 7B
View Activities : [], b.0, b.1, j, p.0, s
View Constraints : Next (One Way)



Skeleton Configuration
Event Log : log7 | split: [n, f], [h, i], [c,i], [h.0, c.0], [p, e], [c.0, h.0.0], 7B
Required Activities Filter : b.1, b.0
View Activities : [], b.0, b.1, f, j, l, n.0, p.0, s, |>
View Constraints : Always After, Always Before

The figure above on the left-hand side shows the *Next* (*One Way*) relations between these activities. Note that *Always* constraints are of no use here, as both implicit choices are optional: *b.1* may be preceded by *b.0*, or may be not, etc. I do not have an explicit constraint yet that says that if both occur, then one always follows the other: If both *b.0* and *b.1* occur in a trace, then *b.1* follows *b.0*. However, if I filter in only those traces where both occur, things become much more clear, as shown in the figure above on the right-hand side. This filtering tactics will be used later on in the classification.

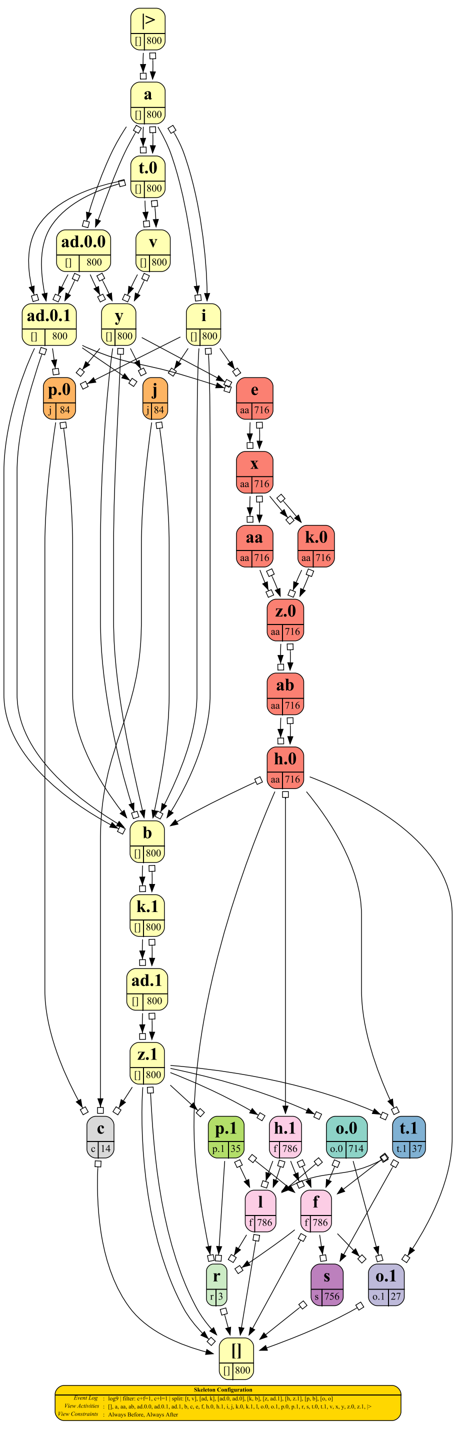
3.2.8 log8



Skeleton Configuration

Event Log	:	log8
View Activities	:	[], a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, |>
View Constraints	:	Always Before, Always After

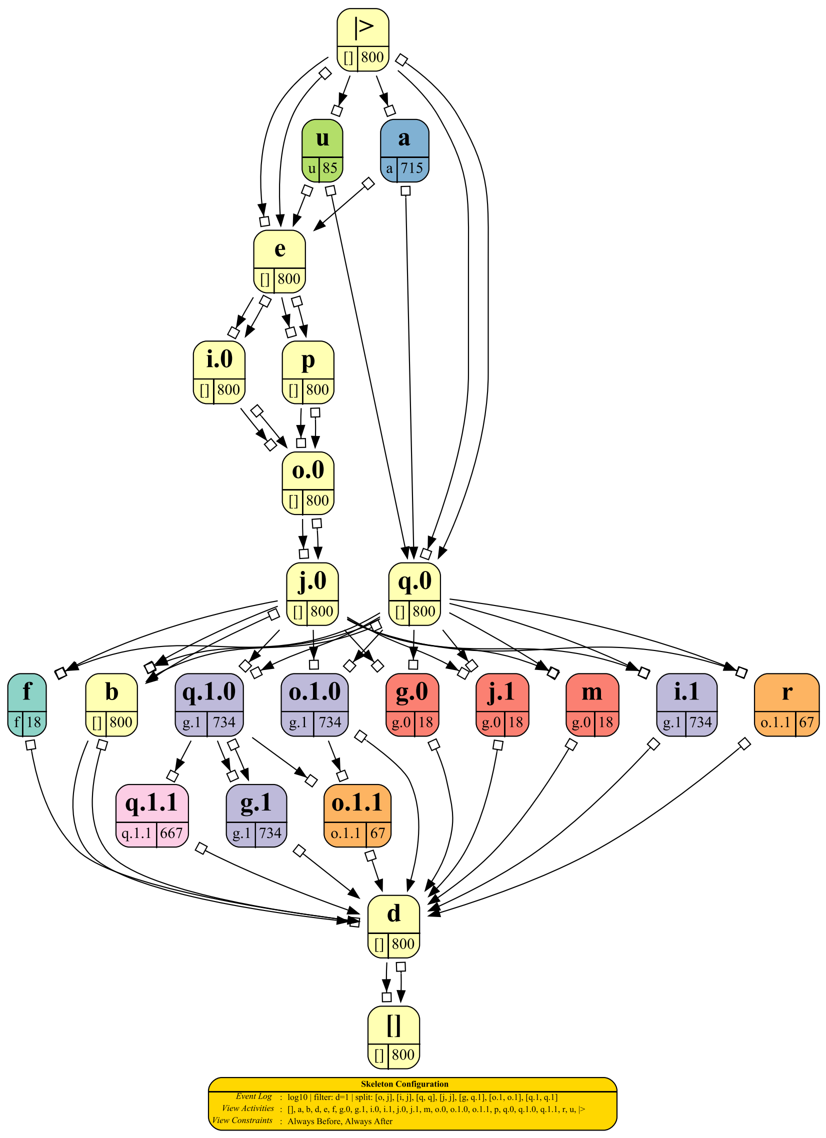
3.2.9 log9



Skeleton Configuration

Event Log	:	log9 | filter: c+f=1, c+l=1 | split: [t, v], [ad, k], [ad.0, ad.0], [k, b], [z, ad.1], [h, z.1], [p, b], [o, o]
View Activities	:	[], a, aa, ab, ad.0.0, ad.0.1, ad.1, b, c, e, f, h.0, h.1, i, j, k.0, k.1, l, o.0, o.1, p.0, p.1, r, s, t.0, t.1, v, x, y, z.0, z.1, |>
View Constraints	:	Always Before, Always After

3.2.10 log10

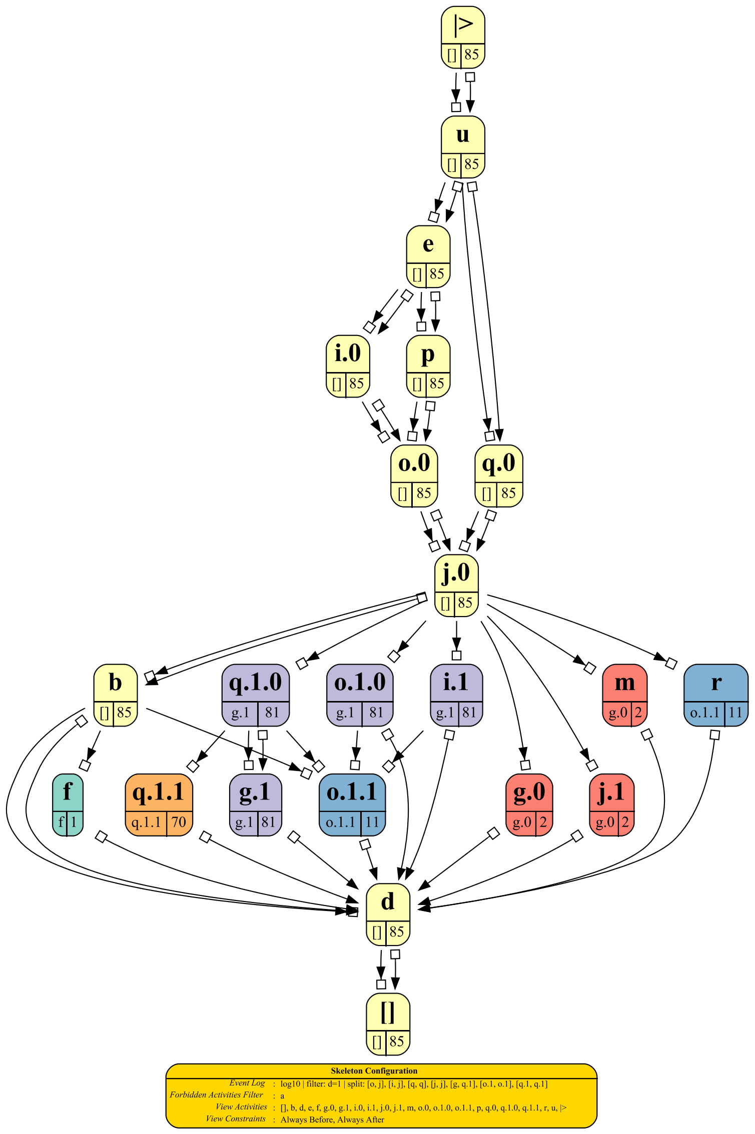

|> [] 800
u u 85
a a 715
e [] 800
i.0 [] 800
p [] 800
o.0 [] 800
j.0 [] 800
q.0 [] 800
f f 18
b [] 800
q.1.0 g.1 734
o.1.0 g.1 734
g.0 g.0 18
j.1 g.0 18
m g.0 18
i.1 g.1 734
r o.1.1 67
q.1.1 q.1.1 667
g.1 g.1 734
o.1.1 o.1.1 67
d [] 800
[] [] 800
Skeleton Configuration
Event Log : log10 | filter: d=1 | split: [o, j], [i, j], [q, q], [j, j], [g, q.1], [o.1, o.1], [q.1, q.1]
View Activities : [], a, b, d, e, f, g.0, g.1, i.0, i.1, j.0, j.1, m, o.0, o.1.0, o.1.1, p, q.0, q.1.0, q.1.1, r, u, |>
View Constraints : Always Before, Always After

4 Classification

The log skeletons as discovered in the previous section will be used to classify the traces in the test logs. To classify the test traces from a test log using some training log, the following procedure C is applied:

1. If a filter is provided for the training log, apply it on the training log to filter out the noise.
2. If a splitter is provided for the training log, apply it on the training log to split (reoccurring) activities.
3. Discover a training log skeleton from the training log.
4. For every test trace in the test log:
   (a) Create a single-trace test log containing only this test trace.
   (b) If a splitter is provided for the training log, apply it on the single-trace test log to split (reoccurring) activities.
   (c) Discover a single-trace test log skeleton from the single-trace test log.
   (d) Check whether all relevant constraints from the training log skeleton are satisfied by the single-trace test log skeleton.
   (e) If not, classify the corresponding trace as negative.

However, there is more to it, as we will not check it on only the discovered model, but also on a number of submodels of that discovered model. Take, for example, *log10* again as example. We already (more or less) concluded that a and u were in a choice construct in the beginning of the model: Either we start with an a, or with a u, and after that both do not occur again. To reach this conclusion, recall that we combined knowledge obtained using different views on the model. There is, however, a simpler way to do this: If we would remove all traces where a does not occur, u would automatically fall 'in-line' with $|>$, e, etc. The following figure shows this result.



Skeleton Configuration

Event Log	:	log10 | filter: d=1 | split: [o, j], [i, j], [q, q], [j, j], [g, q.1], [o.1, o.1], [q.1, q.1]
Forbidden Activities Filter	:	a
View Activities	:	[], b, d, e, f, g.0, g.1, i.0, i.1, j.0, j.1, m, o.0, o.1.0, o.1.1, p, q.0, q.1.0, q.1.1, r, u, |>
View Constraints	:	Always Before, Always After

Clearly, now we can easily check by the *Always Together* constraint whether u is always there if a is not.

Based on this, we use the following procedure C^+ for the classification:

1. Apply procedure C on the training log and the test log, using the three *Always* constraints as relevant constraints.
2. For every activity in the training log:
   (a) Create a first training sublog from the training log that contains all traces that contain the activity (filtering in).
   (b) Create a first test sublog from the test log that contains all traces that contain the activity.
   (c) If we have classified less than 10 traces as negative, apply procedure C on the first training sublog and the first test sublog, using the *Always Together* constraint as the only relevant constraint.

   (d) Create a second training sublog from the training log that contains all traces that do not contain the activity (filtering out).
   (e) Create a second test sublog from the test log that contains all traces that do not contain the activity.
   (f) If we have classified less than 10 traces as negative, apply procedure C on the second training sublog and the second test sublog, using the *Always Together* constraint as the only relevant constraint.

3. Repeat Step 2 but now for every two different activities in the training log, where the first activity can be filtered in or out, and the second can be filtered in our out.
4. Repeat Step 2 but now for every three different activities in the training log, where all three activities will be filtered out.
5. Repeat Step 2, Step 3, and Step 4 but now using both the *Always Before* and *Always After* constraints as relevant constraints.
6. Repeat Step 2, Step 3, and Step 4 but now using both the *Next* (*One Way*) and *Next* (*Two Ways*) constraints as relevant constraints.
7. Classify any unclassified traces as positive.

5 Implementation

The entire approach has been implemented in the *LogSkeleton* package of ProM 6. At the moment of writing, this package is not yet available in the ProM Nightly Build, but it will be made available on July 1st, 2017. If needed, the package can be checked out from our SVN repository[1], after which ProM can be started using the `ProM with UITopia (LogSkeleton)` run configuration in Eclipse. The plug-ins will then be available to the user.

This package contains the following plug-ins (in alphabetical order):

Classify Test Log using Log Skeleton Takes a training event log and a test event log, and returns a log containing the positive traces from the test log. The classification is done by using a collection of log skeletons of the training logs and filtered sublogs of the training log.

Export Log Skeleton Exports a log skeleton to a file (with extension *lsk*).

Filter Event Log on Log Skeleton Creates a sublog from the provided event log. This sublog contains the fitting traces according to the provided log skeleton.

Filter PDC 2017 Event Log Early attempt to filter out the noise from the noisy logs in a generic way. Not used anymore.

Import Log Skeleton Imports a log skeleton from file (with extension *lsk*).

Log Skeleton Browser Visualizes the provided log skeleton, allowing the user to select which activities and/or constraints to show.

Log Skeleton Builder Creates a log skeleton from the provided event log. In this event log, the case identifiers and activity names are stored in the $concept : name$ attributes of the traces and events.

Log Skeleton Filter and Browser Visualizes the provided event log using log skeletons. The visualizer allows the users to set required activities, forbidden activities, and splitters. The visualizer filters the provided event log on the required and forbidden activities as selected by the user, then splits the filtered event log on the splitters as set by the users, then creates a log skeleton from the split filtered event log, and then visualizes this log skeleton using the *Log Skeleton Browser* plug-in.

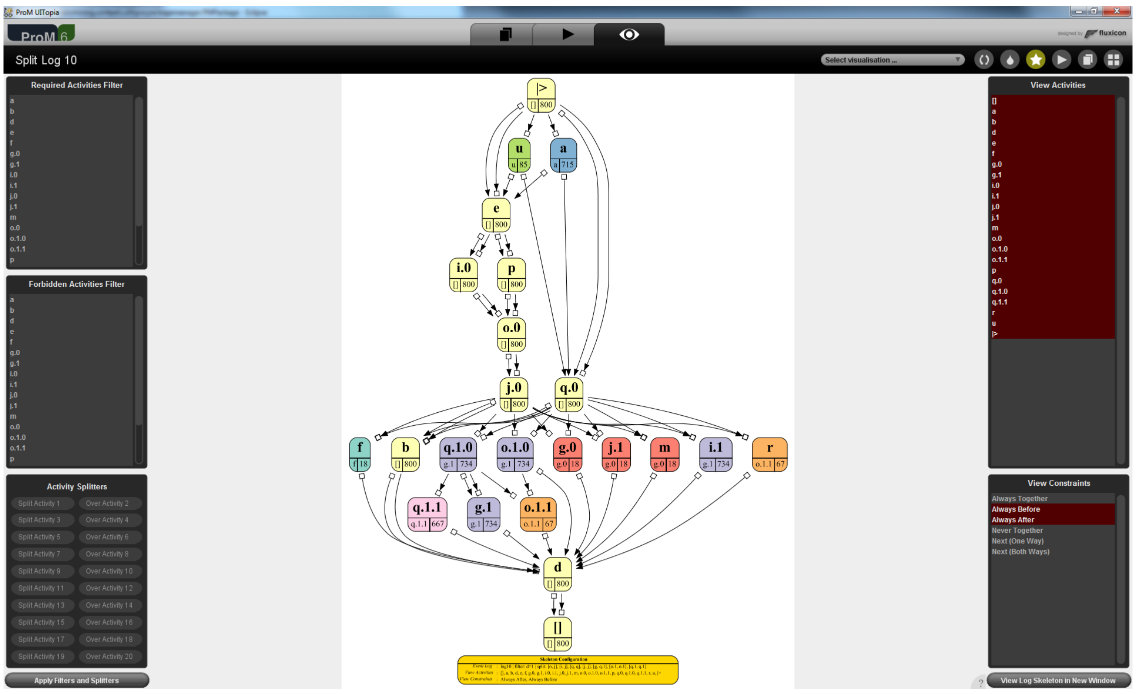
PDC 2017 Log X Filter Creates a filtered sublog from the provided event log X containing those 800 traces that we believe are noise-free. Possible values for X are $1, 2, 5, 9$, and 10.

PDC 2017 Log X Splitter Create an edited log from the provided event log X containing the same traces and number of events, but where (we believe) relevant activities have been split into multiple activities. This can be done to resolve reoccurring activities, but also to unfold loops. Possible values for X are $2, 4, 5, 7, 9$, and 10.

[1]`https://svn.win.tue.nl/repos/prom/Packages/LogSkeleton/Trunk`

PDC 2017 Test Classifies the test logs. Results in an overview for each of the test logs according to the directions as provided by the organizers. This plug-ins assumes that the training log X is located in a specific file[2], the first test log X is located in another specific file[3], etc.

PDC 2017 Test 2016 Classifies the test logs from the 2016 edition of the Process Discovery Contest. Used to check whether the approach made sense. Also assumes fixed locations for the various log files.

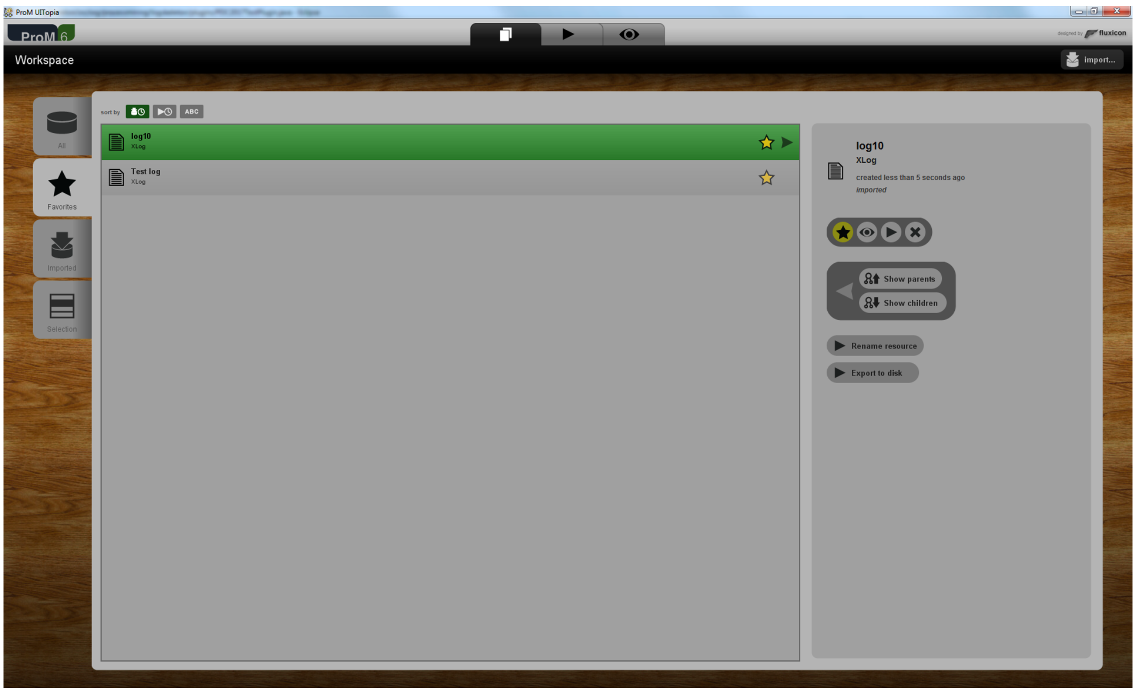


The figure above shows the *Log Skeleton Filter and Browser* on *log10*, after the filter and splitters for this log have been applied. In the middle, the log skeleton is visualized. On the right, the user can select which activities to show and which constraints to show. As soon as the user changes the selection, the visualization in the middle will be updated. Furthermore, the user can decide to open a new window visualizing the current log skeleton. This allows for easy comparison between different log skeletons, if needed. On the left, the user can select activity filters and activity splitters which can be applied to the log before a log skeleton is build for it. The user can select required activities (only traces where all these activities occur will be filtered in) and forbidden activities (only traces where none of these activities occur will be filtered in). Furthermore, the user can enter activity splitters by providing the name of the activity s/he wants to split and the name of the activity it should be split on. The user needs to apply these filters and splitters using the button below, in which case the log will first be filtered, then split, then a log skeleton will be build for it, which will then be visualized to the right. This new visualization will always start with all activities selected, and the *Always Before* and *Always After* constraints selected (note that the *Always Together* constraint is always visualized using the colors). If both the *Always Before* and *Always After* constraints are selected and the *Next* (*One Way*) constraint is not, then the visualizer adds a *Next* (*One Way*) constraint from an activity to another activity if there are no *Always* constraint from that activity to that other activity and if the *Next* (*One Way*) constraint occurs for at least 20% of either the activity or the other activity.

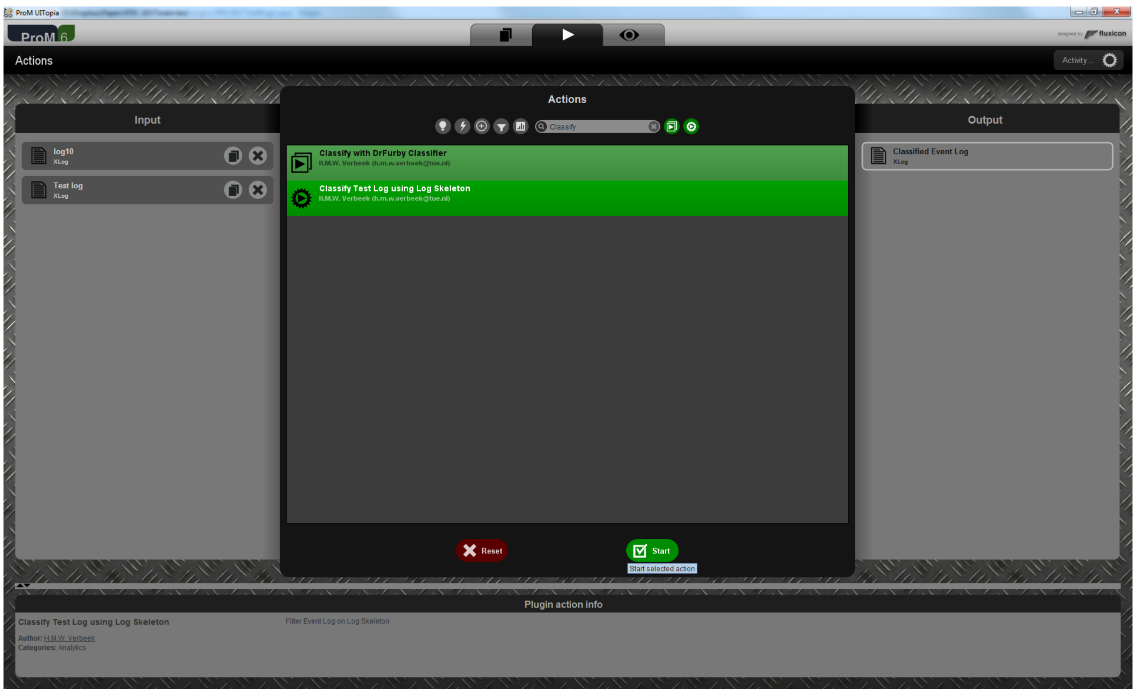
Classifying a test log using a training log is easy. Let's assume you want to classify the test log `test_log_june_10.xes` on the training log `log10.xes`. First, you need to import both logs:

[2]`D:\DropBox\Projects\PDC 2017\log`X`.xes`

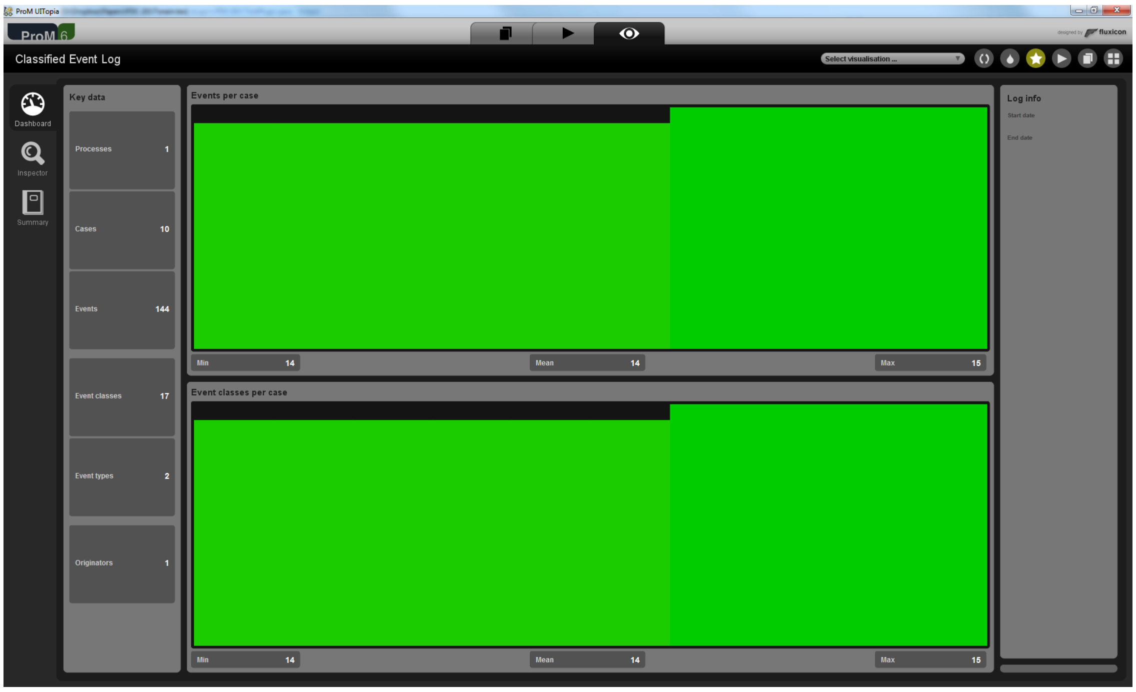
[3]`D:\DropBox\Projects\PDC 2017\test_log_may\test_log_may`X`.xes`



Second, with the training log selected, you start the *Classify Test Log using Log Skeleton* plug-in and add the test log as second argument:


Actions
Input
Actions
Output
log10
Test log
Classified Event Log
Classify with DrFurby Classifier
H.M.W. Verbeek (h.m.w.verbeek@tue.nl)
Classify Test Log using Log Skeleton
H.M.W. Verbeek (h.m.w.verbeek@tue.nl)
Reset
Start
Start selected action
Plugin action info
Classify Test Log using Log Skeleton
Filter Event Log on Log Skeleton
Author: H.M.W. Verbeek
Categories: Analytics

Third, you run the plug-in by selecting the *Start* button. This creates a sub log containing all positive traces from the test log, and automatically shows it:



Classified Event Log

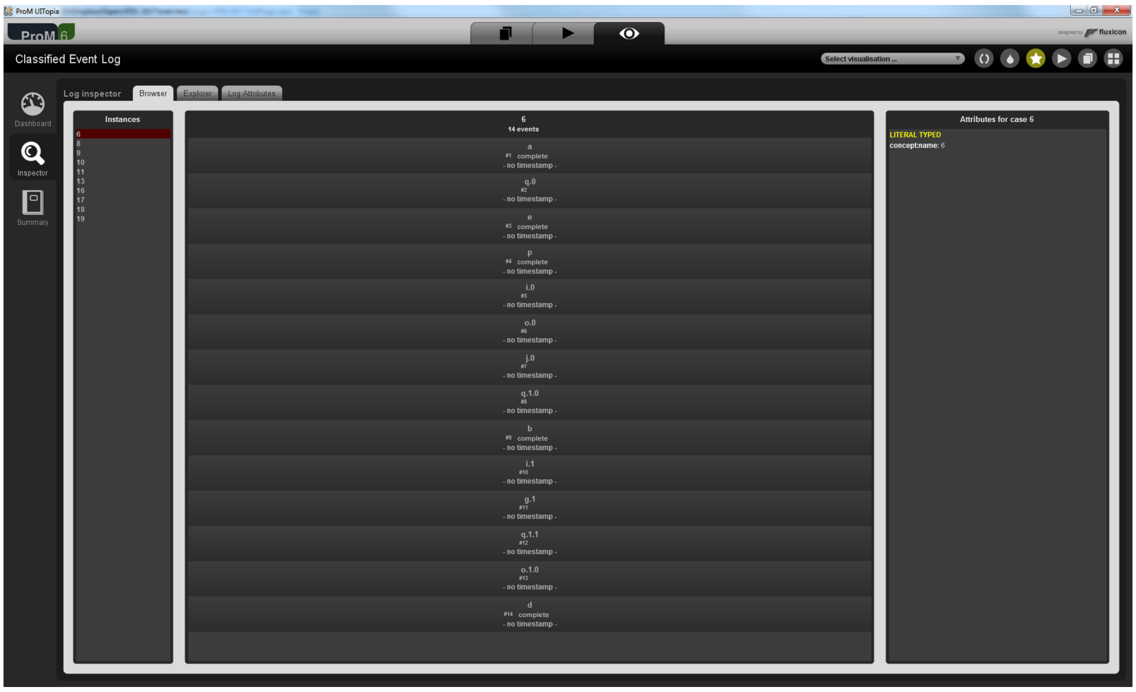
Key data

Processes	1
Cases	10
Events	144
Event classes	17
Event types	2
Originators	1

Events per case: Min 14, Mean 14, Max 15

Event classes per case: Min 14, Mean 14, Max 15

Finally, if you want to know which traces were classified positive, select the *Inspector* tab:



Classified Event Log

Log inspector — Browser

Instances: 6, 8, 9, 10, 11, 13, 16, 17, 18, 19

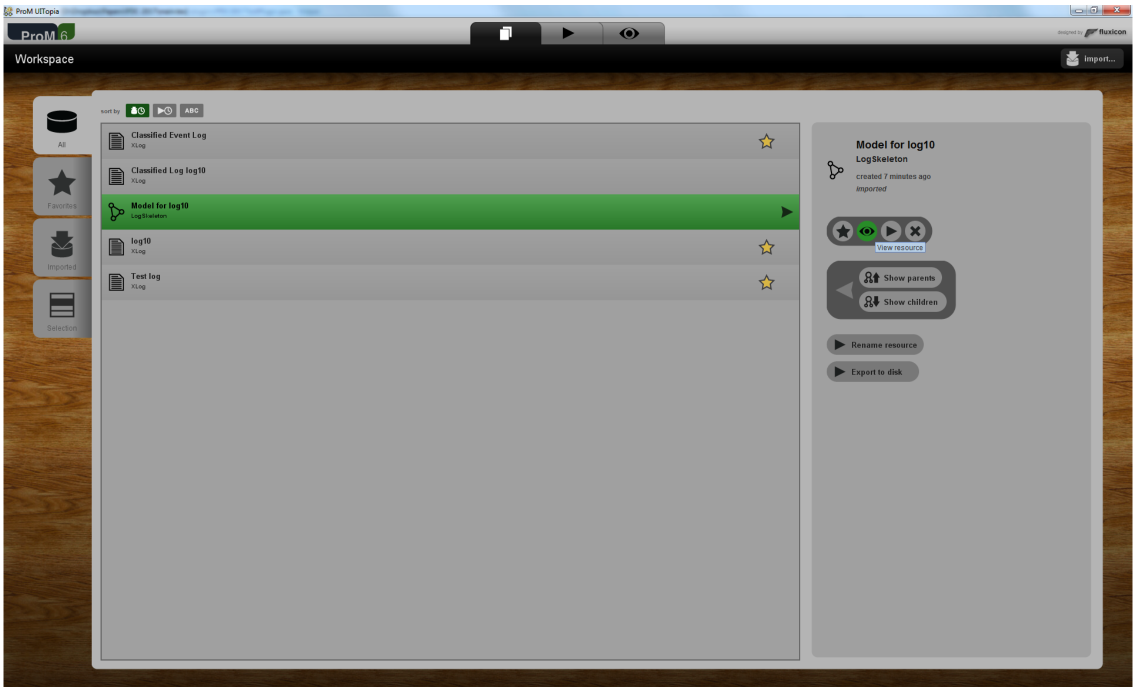
```
6
14 events
a
#1 complete
- no timestamp -
q.0
#2
- no timestamp -
e
#3 complete
- no timestamp -
p
#4 complete
- no timestamp -
i.0
#5
- no timestamp -
o.0
#6
- no timestamp -
j.0
#7
- no timestamp -
q.1.0
#8
- no timestamp -
b
#9 complete
- no timestamp -
i.1
#10
- no timestamp -
g.1
#11
- no timestamp -
q.1.1
#12
- no timestamp -
o.1.0
#13
- no timestamp -
d
#14 complete
- no timestamp -
```

Attributes for case 6

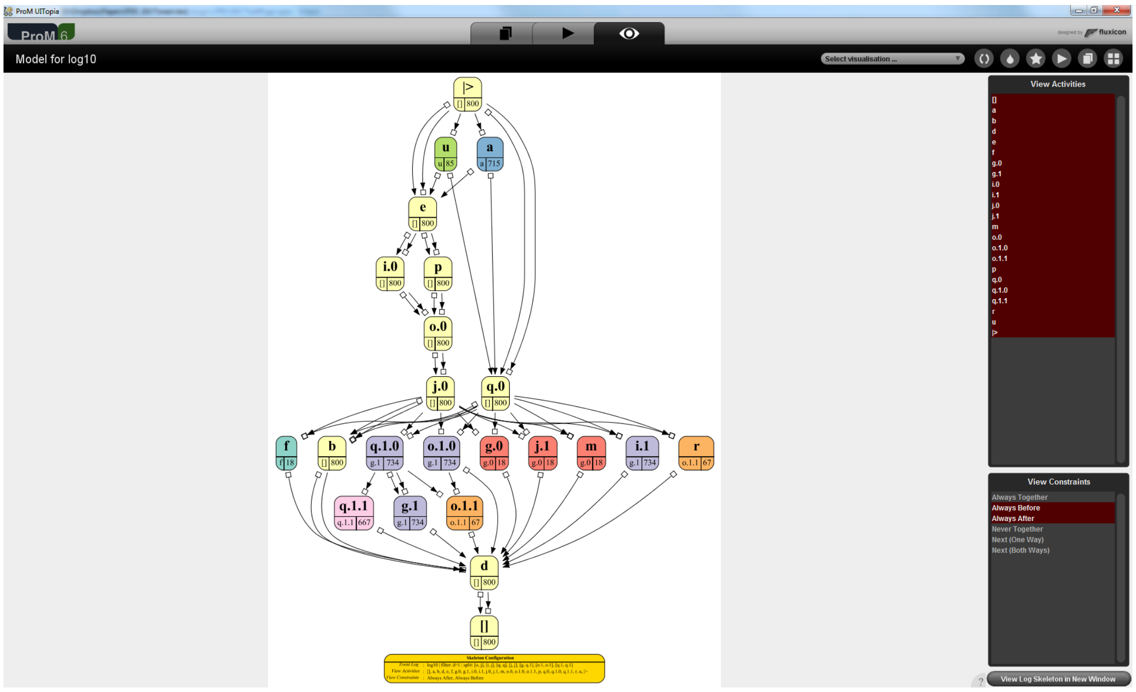
LITERAL TYPED
concept:name: 6

This shows that the traces 6, 8, 9, 10, 11, 13, 16, 17, 18, and 19 were classified positive.

As a side effect of the classification, the main log skeleton has been made available in ProM. To access this log skeleton, select the workspace tab in ProM, and then the *All* tab:



You can view the generated log skeleton by selecting it and then by selecting the *View resource* button (with the *eye* icon):



Please note that the actual classification differs per event log:

- Different filters are used for different event logs.
- Different splitter are used for different event logs.
- For some event logs, some test traces are added to improve (hopefully) the classification.

For this, it is important that the names of the logs are *log1*, …, *log10*. Based on the name of the log, as captured by the $concept : name$ attribute of the log, a different filter is applied, a different splitter is applied, and possibly some extra traces are added.

6 Results

6.1 May Test Logs

	model_1	model_2	model_3	model_4	model_5	model_6	model_7	model_8	model_9	model_10
trace_1	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE
trace_2	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
trace_3	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE
trace_4	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE
trace_5	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE
trace_6	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE
trace_7	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE
trace_8	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE
trace_9	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE
trace_10	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE
trace_11	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
trace_12	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE
trace_13	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE
trace_14	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
trace_15	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
trace_16	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE
trace_17	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE
trace_18	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
trace_19	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
trace_20	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
#True	10	10	10	10	10	10	10	10	10	10

It has been confirmed by the organizers that this classification is 100% correct.

6.1.1 Test Log 1

```
Case 18: Always Together fails for [[], c, g, m, p, s, t, u, v, w, |>]
Case 14: Always Together fails for [d, h, o]
Case 6: Always Before fails for q, missing are [e]
Case 10: Always Together fails for [[], c, g, m, p, s, t, u, v, w, |>]
Case 1: Always Together fails for [[], c, g, m, p, s, t, u, v, w, |>]
Case 7: Always Before fails for n, missing are [g, u, t]
Case 15: Always Together fails for [d, h, o]
Case 19: Always Together fails for [b, l]
Case 13: Always After fails for e, missing are [o, h]
Case 19: Always Together fails for [[], c, g, m, p, s, t, u, v, w, |>]

Case 8 excluded by positive filter [a] and negative filter [], support = 717
Case 8: Always Before fails for i, missing are [e]
```

6.1.2 Test Log 2

```
Case 20: Always After fails for l, missing are [m]
Case 10: Always Together fails for [[], a.0, d, k, m, o, s.0, |>]
Case 10: Always Together fails for [a.1, n, s.1, t]
Case 7: Always Before fails for p, missing are [j]
Case 18: Always Together fails for [a.1, n, s.1, t]
Case 1: Always Together fails for [h, j, p]
Case 15: Always After fails for p, missing are [i]
Case 18: Always Together fails for [[], a.0, d, k, m, o, s.0, |>]
Case 11: Always Together fails for [[], a.0, d, k, m, o, s.0, |>]
Case 8: Always Together fails for [a.1, n, s.1, t]
Case 19 excluded by positive filter [] and negative filter [e], support = 603
Case 19: Always Together fails for [b, f, g, i]

Case 9 excluded by positive filter [a.0] and negative filter [], support = 800
Case 9: Next fails for [t, s.1]
```

6.1.3 Test Log 3

```
Case 12: Always Together fails for [[], h, |>]
Case 18: Always Before fails for q, missing are [g]
Case 7: Always Together fails for [c, g, m, q, u]
Case 4: Always Together fails for [c, g, m, q, u]
Case 2: Always Together fails for [c, g, m, q, u]
Case 8: Always Together fails for [c, g, m, q, u]
Case 7: Always Together fails for [[], h, |>]
Case 11: Always Together fails for [c, g, m, q, u]

Case 13 excluded by positive filter [] and negative filter [a], support = 117
Case 13: Always Together fails for [[], h, p, s, |>]

Case 15 excluded by positive filter [] and negative filter [a], support = 117
Case 15: Always Together fails for [[], h, p, s, |>]

Case 9 excluded by positive filter [] and negative filter [d], support = 933
Case 9: Always Together fails for [c, g, m, q, r, u]
```

6.1.4 Test Log 4

```
Case 12: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 7: Always Together fails for [d, r.0]
Case 2: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 8: Always After fails for a, missing are [m.0]
Case 6: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 3: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 15: Always After fails for f, missing are [b]
Case 10: Always Together fails for [d, r.0]
Case 7: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 5: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]

Case 13 excluded by positive filter [] and negative filter [p], support = 871
Case 13: Always Together fails for [a, m.0]
```

6.1.5 Test Log 5

```
Case 12: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 2: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 20: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 6: Always Before fails for f, missing are [u, l, j]
Case 19: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 9: Always After fails for n, missing are [v]
Case 1: Always After fails for b, missing are [c]

Case 8 excluded by positive filter [g.1.0] and negative filter [], support = 54
Case 8: Always Together fails for [n, q]

Case 10 excluded by positive filter [] and negative filter [m], support = 544
Case 10: Always Together fails for [g.1.0, i.1.0]

Case 18 excluded by positive filter [n, a.1] and negative filter [], support = 245
Case 18: Always Together fails for [k, q]
```

6.1.6 Test Log 6

```
Case 15: Always Before fails for p, missing are [h]
Case 8: Always After fails for d, missing are [k]
Case 7: Always Together fails for [h, p]
Case 6: Always Together fails for [h, p]
Case 1: Always Together fails for [h, p]
Case 11: Always Together fails for [h, p]
Case 3: Always Together fails for [h, p]
Case 2: Always Together fails for [h, p]
Case 10: Always After fails for p, missing are [k]

Case 13 excluded by positive filter [] and negative filter [a], support = 505
Case 13: Always Together fails for [[], b, g, j, |>]
```

6.1.7 Test Log 7

```
Case 10: Always Together fails for [[], f, j, l, n.0, |>]
Case 7: Always Together fails for [[], f, j, l, n.0, |>]
Case 12: Always Together fails for [[], f, j, l, n.0, |>]
Case 15: Always Before fails for p.0, missing are [j]
Case 6: Always Together fails for [[], f, j, l, n.0, |>]
Case 4: Always Together fails for [[], f, j, l, n.0, |>]
Case 8: Always Before fails for j, missing are [n.0]
Case 3: Always Together fails for [[], f, j, l, n.0, |>]

Case 11 excluded by positive filter [] and negative filter [b.1, n.1], support = 111
Case 11: Always Together fails for [[], f, j, l, n.0, s, |>]

Case 14 excluded by positive filter [] and negative filter [b.1, n.1], support = 111
Case 14: Always Together fails for [[], f, j, l, n.0, s, |>]
```

6.1.8 Test Log 8

```
Case 18: Always Together fails for [a, j]
Case 18: Always Together fails for [[], d, |>]
Case 14: Always Together fails for [a, j]
Case 19: Always Before fails for h, missing are [d]
Case 16: Always Together fails for [[], d, |>]
Case 18: Always Together fails for [b, q]
Case 15: Always Together fails for [b, q]

Case 6 excluded by positive filter [b, i] and negative filter [], support = 121
Case 6: Always Together fails for [[], b, d, h, i, o, p, q, |>]

Case 13 excluded by positive filter [n] and negative filter [b], support = 68
Case 13: Always Together fails for [[], d, k, l, n, |>]

Case 11 excluded by positive filter [] and negative filter [f, e, c], support = 64
Case 11: Always Together fails for [[], d, m, |>]

Case 17 excluded by positive filter [] and negative filter [f, e, c], support = 64
Case 17: Always Together fails for [[], d, m, |>]

Case 20 excluded by positive filter [] and negative filter [g, a, i], support = 80
Case 20: Always Together fails for [[], d, n, |>]
```

Note that for this log the *Next* constraints play a role, as 3 out of 10 non-fitting traces (11, 17, and 20) were the result of these constraints.

6.1.9 Test Log 9

```
Case 13: Always Before fails for o.0, missing are [z.1]
Case 12: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
Case 15: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
Case 15: Always Together fails for [aa, ab, e, h.0, k.0, x, z.0]
Case 17: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
Case 4: Always After fails for t.1, missing are [s]
Case 15: Always Together fails for [f, h.1, l]
Case 9: Always Before fails for s, missing are [f]
Case 5: Always After fails for h.0, missing are [h.1]
Case 3: Always Before fails for r, missing are [p.1]
Case 18: Always Together fails for [aa, ab, e, h.0, k.0, x, z.0]
Case 15: Always Together fails for [j, p.0]

Case 11 excluded by positive filter [] and negative filter [s], support = 44
Case 11: Always Together fails for [o.0, o.1]
```

6.1.10 Test Log 10

```
Case 15: Always Before fails for b, missing are [j.0]
Case 11: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 7: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 4: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 1: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 19: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 9: Always After fails for i.1, missing are [d]
Case 12: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 8: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 12: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 1: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 7: Always Together fails for [g.0, j.1, m]
Case 19: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 10: Always Together fails for [g.0, j.1, m]
Case 11: Always Together fails for [g.0, j.1, m]
Case 10: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
```

6.2 June Test Logs

	model_1	model_2	model_3	model_4	model_5	model_6	model_7	model_8	model_9	model_10
trace_1	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
trace_2	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE
trace_3	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE
trace_4	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE
trace_5	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE
trace_6	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
trace_7	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE
trace_8	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
trace_9	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE
trace_10	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE
trace_11	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE
trace_12	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
trace_13	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE
trace_14	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE
trace_15	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	FALSE
trace_16	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE
trace_17	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE
trace_18	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
trace_19	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
trace_20	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE
#True	10	10	10	10	10	10	10	10	10	10

It has been confirmed by the organizers that this classification is 100% correct for all models except 6. For the model 6, we ran out of our two classification attempts. Best results from the two classification results were 4 misclassifications for model 6. In the end, this would be a score of 196 out of 200. The classification shown above is attempt 3. We hope it will do better than 196 out of 200, but we do not know.

6.2.1 Test Log 1

```
Case 12: Always After fails for a, missing are [d, h]
Case 18: Always Together fails for [d, h, o]
Case 17: Always Together fails for [[], c, g, m, p, s, t, u, v, w, |>]
Case 5: Always Together fails for [[], c, g, m, p, s, t, u, v, w, |>]
Case 17: Always Together fails for [b, l]
Case 6: Always Before fails for o, missing are [d]
Case 9: Always Together fails for [d, h, o]
Case 3: Always Together fails for [[], c, g, m, p, s, t, u, v, w, |>]
Case 2: Always Together fails for [[], c, g, m, p, s, t, u, v, w, |>]
Case 16: Always Before fails for n, missing are [a]
Case 13: Always Before fails for u, missing are [v]
```

6.2.2 Test Log 2

```
Case 10: Always Before fails for k, missing are [m]
Case 13: Always Together fails for [a.1, n, s.1, t]
Case 15: Always Together fails for [a.1, n, s.1, t]
Case 1: Always Together fails for [a.1, n, s.1, t]
Case 2: Always Before fails for r, missing are [o]
Case 8: Always Together fails for [h, j, p]
Case 12: Always Together fails for [h, j, p]
Case 11: Always Before fails for j, missing are [h]

Case 6 excluded by positive filter [] and negative filter [l], support = 396
Case 6: Always Together fails for [[], a.0, d, k, m, o, r, s.0, |>]

Case 4 excluded by positive filter [a.0] and negative filter [], support = 800
Case 4: Next fails for [t, s.1]
```

6.2.3 Test Log 3

```
Case 7: Always Before fails for f, missing are [h]
Case 4: Always Before fails for f, missing are [h]
Case 12: Always Before fails for m, missing are [q]
Case 15: Always Together fails for [c, g, m, q, u]
Case 5: Always Before fails for q, missing are [g]
Case 10: Always Together fails for [c, g, m, q, u]
Case 8: Always Together fails for [b, f, j]
Case 1: Always Before fails for m, missing are [q]
Case 13: Always Before fails for k, missing are [c]
Case 9: Always Before fails for l, missing are [c]
```

6.2.4 Test Log 4

```
Case 12: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 19: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 18: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 12: Always Together fails for [d, r.0]
Case 9: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 8: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 10: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 15: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]

Case 1 excluded by positive filter [] and negative filter [a], support = 188
Case 1: Always Together fails for [m.0, p]

Case 16 excluded by positive filter [a] and negative filter [], support = 812
Case 16: Next fails for [e, o]

Case 20 excluded by positive filter [a] and negative filter [], support = 812
Case 20: Next fails for [v, m.0]
```

6.2.5 Test Log 5

```
Case 14: Always Before fails for k, missing are [v, b, r]
Case 6: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 10: Always Together fails for [t, w]
Case 4: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 3: Always Together fails for [o, s]
Case 7: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 1: Always After fails for w, missing are [c]
Case 5: Always Before fails for u, missing are [c]

Case 18 excluded by positive filter [] and negative filter [n, a.1], support = 72
Case 18: Always Together fails for [[], a.0, b, d, f, h, j, k, l, o, p, r, s, u, v, |>]

Case 9 excluded by positive filter [g.0, o] and negative filter [], support = 76
Case 9: Always After fails for m, missing are [g.0]
```

6.2.6 Test Log 6

```
Case 20: Always Together fails for [h, p]
Case 17: Always After fails for f, missing are [k]
Case 18: Always Before fails for p, missing are [h]
Case 15: Always After fails for f, missing are [k]
Case 19: Always After fails for f, missing are [k]

Case 9 excluded by positive filter [] and negative filter [f, n, l], support = 749
Case 9: Always Together fails for [k, s]

Case 10 excluded by positive filter [] and negative filter [f, n, l], support = 749
Case 10: Always Together fails for [k, s]

Case 14 excluded by positive filter [d] and negative filter [], support = 232
Case 14: Always Before fails for f, missing are [d]

Case 8 excluded by positive filter [a] and negative filter [], support = 499
Case 8: Next fails for [e, n]

Case 16 excluded by positive filter [a] and negative filter [], support = 499
Case 16: Next fails for [e, p]
```

6.2.7 Test Log 7

```
Case 2: Always Together fails for [[], f, j, l, n.0, |>]
Case 15: Always Together fails for [[], f, j, l, n.0, |>]
Case 10: Always Together fails for [[], f, j, l, n.0, |>]
Case 6: Always Before fails for p.0, missing are [f]
Case 16: Always Before fails for p.0, missing are [l]
Case 8: Always Before fails for c.0.0, missing are [f, l, j]
Case 17: Always Together fails for [[], f, j, l, n.0, |>]
Case 4: Always Together fails for [[], f, j, l, n.0, |>]
Case 1: Always Together fails for [[], f, j, l, n.0, |>]

Case 13 excluded by positive filter [] and negative filter [b.0], support = 831
Case 13: Next fails for [j, s]
```

6.2.8 Test Log 8

```
Case 8: Always Before fails for p, missing are [h]
Case 1: Always Before fails for q, missing are [b]
Case 12: Always Before fails for b, missing are [d]
Case 14: Always Together fails for [b, q]
Case 7: Always Before fails for q, missing are [b]
Case 3: Always After fails for n, missing are [d]
Case 15: Always Together fails for [[], d, |>]

Case 4 excluded by positive filter [e] and negative filter [b], support = 92
Case 4: Always Together fails for [[], d, e, k, l, |>]

Case 11 excluded by positive filter [g] and negative filter [b], support = 62
Case 11: Always Together fails for [[], d, g, k, l, |>]

Case 20 excluded by positive filter [b] and negative filter [], support = 573
Case 20: Always Before fails for h, missing are [q]
```

6.2.9 Test Log 9

```
Case 4: Always Before fails for ab, missing are [z.0]
Case 3: Always Before fails for r, missing are [h.0, p.1]
Case 10: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
Case 9: Always Together fails for [aa, ab, e, h.0, k.0, x, z.0]
Case 6: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
Case 5: Always After fails for o.0, missing are [f, l]
Case 12: Always Before fails for l, missing are [h.1]
Case 9: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
Case 13: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
Case 9: Always Together fails for [f, h.1, l]
Case 10: Always Together fails for [aa, ab, e, h.0, k.0, x, z.0]
Case 7: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
Case 2: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
```

6.2.10 Test Log 10

```
Case 15: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 2: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 20: Always Together fails for [g.0, j.1, m]
Case 12: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 7: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 15: Always Together fails for [g.0, j.1, m]
Case 14: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 3: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 20: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 5: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 4: Always Together fails for [g.1, i.1, o.1.0, q.1.0]

Case 1 excluded by positive filter [] and negative filter [a], support = 85
Case 1: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, u, |>]
```

6.3 Final Test Logs

	model_1	model_2	model_3	model_4	model_5	model_6	model_7	model_8	model_9	model_10
trace_1	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE
trace_2	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE
trace_3	TRUE	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE
trace_4	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
trace_5	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
trace_6	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
trace_7	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
trace_8	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE
trace_9	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
trace_10	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE
trace_11	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE
trace_12	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
trace_13	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE
trace_14	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
trace_15	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE
trace_16	TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
trace_17	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
trace_18	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE
trace_19	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
trace_20	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
#True	10	10	10	10	10	10	10	10	10	10

It has been confirmed by the organizers that this classification is 100% correct.

6.3.1 Test Log 1

```
Case 2: Always After fails for e, missing are [d, h]
Case 6: Always Together fails for [d, h, o]
Case 4: Always Before fails for n, missing are [a]
Case 8: Always Together fails for [[], c, g, m, p, s, t, u, v, w, |>]
Case 7: Always Before fails for t, missing are [v]
Case 1: Always Together fails for [d, h, o]
Case 9: Always Together fails for [d, h, o]
Case 20: Always Together fails for [d, h, o]
Case 19: Always Together fails for [d, h, o]
Case 13: Always Before fails for n, missing are [u, t]
```

6.3.2 Test Log 2

```
Case 9: Always Before fails for o, missing are [s.0]
Case 20: Always Together fails for [[], a.0, d, k, m, o, s.0, |>]
Case 17: Always Together fails for [a.1, n, s.1, t]
Case 6: Always Together fails for [a.1, n, s.1, t]
Case 2: Always Before fails for l, missing are [o]
Case 12: Always Together fails for [h, j, p]
Case 8: Always Before fails for m, missing are [o]
Case 1: Always After fails for p, missing are [i]

Case 13 excluded by positive filter [] and negative filter [c], support = 605
Case 13: Always Together fails for [b, f, g, q]

Case 11 excluded by positive filter [a.1] and negative filter [], support = 406
Case 11: Next fails for [t, t]
```

6.3.3 Test Log 3

```
Case 18: Always Together fails for [c, g, m, q, u]
Case 19: Always Together fails for [b, f, j]
Case 16: Always Before fails for i, missing are [c]
Case 7: Always Together fails for [c, g, m, q, u]
Case 15: Always Before fails for u, missing are [q]
Case 8: Always Before fails for q, missing are [g]
Case 1: Always After fails for l, missing are [g]
Case 20: Always Before fails for q, missing are [g]
Case 11: Always After fails for l, missing are [g]

Case 5 excluded by positive filter [] and negative filter [d], support = 933
Case 5: Always Together fails for [c, g, m, q, r, u]
```

6.3.4 Test Log 4

```
Case 19: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 20: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 8: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 15: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 3: Always Together fails for [[], b, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]
Case 8: Always Together fails for [d, r.0]

Case 18 excluded by positive filter [] and negative filter [g], support = 910
Case 18: Always Together fails for [e, v]

Case 11 excluded by positive filter [] and negative filter [p], support = 871
Case 11: Always Together fails for [a, m.0]

Case 13 excluded by positive filter [e] and negative filter [v], support = 26
Case 13: Always Together fails for [[], b, e, g, h, i, l, m.1, n, r.1, s, t.0, t.1, u, w, |>]

Case 6 excluded by positive filter [] and negative filter [k], support = 843
Case 6: Always Before fails for e, missing are [m.0]

Case 1 excluded by positive filter [a] and negative filter [], support = 812
Case 1: Next fails for [|>, m.0]
```

6.3.5 Test Log 5

```
Case 2: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 18: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 14: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 5: Always After fails for v, missing are [c]
Case 7: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]
Case 15: Always Before fails for w, missing are [c]
Case 8: Always Together fails for [[], a.0, b, d, f, h, j, l, r, u, v, |>]

Case 4 excluded by positive filter [] and negative filter [e], support = 447
Case 4: Always Together fails for [[], a.0, b, d, f, h, j, k, l, r, u, v, |>]

Case 16 excluded by positive filter [g.1.0] and negative filter [], support = 54
Case 16: Always Together fails for [n, q]

Case 10 excluded by positive filter [n, a.1] and negative filter [], support = 245
Case 10: Always Together fails for [k, q]
```

6.3.6 Test Log 6

```
Case 14: Always Before fails for t, missing are [c]
Case 18: Always After fails for l, missing are [k]
Case 3: Always After fails for d, missing are [k]
Case 13: Always Together fails for [c, t]
Case 6: Always Together fails for [h, p]
Case 19: Always Before fails for p, missing are [h]
Case 15: Always Together fails for [h, p]
Case 20: Always After fails for d, missing are [k]
Case 1: Always Before fails for p, missing are [h]

Case 12 excluded by positive filter [] and negative filter [a], support = 505
Case 12: Always Together fails for [[], b, g, j, |>]
```

6.3.7 Test Log 7

```
Case 2: Always Together fails for [[], f, j, l, n.0, |>]
Case 18: Always Together fails for [[], f, j, l, n.0, |>]
Case 16: Always Before fails for c.0.0, missing are [j]
Case 6: Always Together fails for [[], f, j, l, n.0, |>]
Case 17: Always Before fails for h.0.0, missing are [l]
Case 19: Always Before fails for p.0, missing are [f]

Case 11 excluded by positive filter [] and negative filter [b.1, n.1], support = 111
Case 11: Always Together fails for [[], f, j, l, n.0, s, |>]

Case 20 excluded by positive filter [] and negative filter [b.1, n.1], support = 111
Case 20: Always Together fails for [[], f, j, l, n.0, s, |>]

Case 9 excluded by positive filter [n.1] and negative filter [], support = 516
Case 9: Always After fails for p.0, missing are [n.1]

Case 15 excluded by positive filter [] and negative filter [b.0], support = 831
Case 15: Next fails for [l, s]
```

6.3.8 Test Log 8

```
Case 19: Always Together fails for [a, j]
Case 9: Always Before fails for q, missing are [b]
Case 20: Always Before fails for c, missing are [d]
Case 4: Always Before fails for q, missing are [b]
Case 18: Always Together fails for [h, o, p]
Case 17: Always Before fails for c, missing are [d]
Case 11: Always Before fails for q, missing are [b]
Case 5: Always Together fails for [[], d, |>]

Case 7 excluded by positive filter [g, b] and negative filter [], support = 142
Case 7: Always Together fails for [e, l]

Case 10 excluded by positive filter [g, b] and negative filter [], support = 142
Case 10: Always Together fails for [e, l]
```

6.3.9 Test Log 9

```
Case 9: Always Before fails for t.1, missing are [h.0]
Case 2: Always Together fails for [j, p.0]
Case 16: Always Before fails for l, missing are [h.1]
Case 6: Always After fails for o.0, missing are [f]
Case 13: Always Together fails for [f, h.1, l]
Case 11: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]
Case 3: Always After fails for o.0, missing are [l]
Case 10: Always After fails for t.1, missing are [s]
Case 1: Always Together fails for [[], a, ad.0.0, ad.0.1, ad.1, b, i, k.1, t.0, v, y, z.1, |>]

Case 7 excluded by positive filter [] and negative filter [s], support = 44
Case 7: Always Together fails for [p.1, r]
```

6.3.10 Test Log 10

```
Case 13: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 2: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 6: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 13: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 6: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 13: Always Together fails for [o.1.1, r]
Case 4: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 13: Always Together fails for [g.0, j.1, m]
Case 8: Always Before fails for q.1.0, missing are [j.0]
Case 3: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, |>]
Case 5: Always Together fails for [g.0, j.1, m]
Case 5: Always Together fails for [g.1, i.1, o.1.0, q.1.0]
Case 10: Always Together fails for [g.0, j.1, m]
Case 10: Always Together fails for [g.1, i.1, o.1.0, q.1.0]

Case 1 excluded by positive filter [] and negative filter [a], support = 85
Case 1: Always Together fails for [[], b, d, e, i.0, j.0, o.0, p, q.0, u, |>]

Case 9 excluded by positive filter [] and negative filter [o.1.1], support = 733
Case 9: Always Together fails for [g.1, i.1, o.1.0, q.1.0, q.1.1]
```

7 Concluding Remarks

This document has detailed my contribution to the Process Discovery Contest of 2017. My contribution uses a novel concept called *log skeletons*. In a way, one could think of a log skeleton as an X-ray view of an event log. Using a log skeleton, a number of constraints (as detected in the event log) are made prominent. The most important constraints are the *Always* constraints: the *Always Together* constraint, the *Always Before* constraint, and the *Always After* constraint.

An *Always Together* constraint between two activities indicates that these activities occur equally often in every trace of the event log. Especially in combination with the artificial start (| >) and end ([]) activities, this is a very usable constraint, which allows me to detect about 70% of the non-fitting traces.

An *Always Before* constraint between two activities indicates that an occurrence of the target activity in a trace is always preceded in that trace by an occurrence of the source activity, whereas an *Always After* constraint indicates that an occurrence of the source activity is always followed by an occurrence of the target activity. About 25% of the non-fitting traces were detected because they violated these constraints.

The remaining 5% of the non-fitting traces were detected using a *Next* constraint, that is, using the directly-follows graph.

Currently, the log skeletons work best for event logs that contain no noise. For the Process Discovery Contest 2017, the log skeletons work because I was able to filter out the noisy traces from the noisy event logs. Nevertheless, I would like to stress that to create the actual filters for the noisy event logs, I did use the noisy log skeletons of the noisy logs. Based on what I saw in the noisy log skeleton, I created the log filters. Of course, for this I used the fact that the organizers told the contestants which logs contained noise, and what the type of noise was. For event logs with arbitrary noise, this would have bene much harder, and perhaps I need to extend the current log skeletons with noise someday. An *Always Together* constraint would then not be on every trace, but on, say, 95% of all traces.

The organizers also informed us which event logs contained reoccurring activities, but of course, the event logs also informed us of this: The event logs for which the activity names were not consecutive, contained reoccurring activities. The latter shows (I believe) that the organizers first created an event log without reoccurring activities, and then renamed some activities to already existing activities, hereby creating the reoccurring activities. Knowing this, one knows approximately how many reoccurring activities to expect (as many as there are 'holes' in the order of activity names). Using the *Log Skeleton Filter and Browser*, and by using a trial-and-error approach, I created the *splitters* that undo these renamings by the organizers as good as possible. Like with the filters, of course, I could be wrong here, and could have created the wrong filters and the wrong splitters, but I believe I'm not that far off the mark.

Finally, participating in the Process Discovery Contest 2017 was fun, and gave me the idea for the log skeletons, but did cost me time. Time I may have spend on other things which were also important (but less fun, I guess). For that reason I hereby volunteer myself to participate in the organization of next year's challenge, if needed. That will prevent me from participating in next's year's challenge, which, in the end, will save me time.

A Change Log

Version 1.1 Added results on final test logs (see Section 6.3).

Version 1.0 Original report.