Case study on Process Mining with PN - Alemanno

February 24, 2023

EXAM: FORMAL METHODS

Prof. De Carolis Berardina

Case Study: Process mining with PN

Alemanno Giuseppe Matr. 789750 g.alemanno11@studenti.uniba.it

Academic Year 2022-2023

1 Introduction to Process Mining

Process Mining aims to support the understanding and improvement of real processes by extracting knowledge from event logs. For the event log to be used for process mining techniques, it must at least contain cases and activities. Each case represents a process and is usually marked in the table by a numeric or text id.
The activity is an element that is monitored within the individual processes.
A frequently used, but not essential, component is the timestamp, which allow us to determine the order and duration of events.

There are three general types of process mining:

- **Process Discovery:** It consists of techniques that generate a process model based on previously recorded event logs. Nowadays, many algorithms exist that given logs as input return a model. This model is then used to visualize the process. Although generation is now automatic, it is necessary to ensure that quality models are obtained. The measurement of model quality can be defined by the following quality criteria:
  - *Generalization* – the model should generalize the behavior present in the event log.
  - *Precision* – the model should not allow a behavior unrelated to the one stored in the log.
  - *Fitness* – the model should allow the behavior present in the event log.
  - *Simplicity* – the model should be as simple as possible. There is an inverse relationship between Generalization and Precision so these criteria compete among them.

- **Process Conformance:** It consists of techniques that, given an existing process model and event logs, allow to detect deviations between the two.
- **Process Enhancement:** It consists of techniques whose aim is to improve or extend an already existing process model.

2 Goal of the case study

The objective of the case study is to analyze a dataset of human actions performed in an smart home environment. Then apply valid algorithms in order to obtain a quality process model which help increase our understanding of the real process. Then test the models with the original and the infrequent version of logs to compare them and understand their limitations and potential.

3 Dataset

In order to find a valid dataset, I consulted several sites offering open data from different companies and research realities, such as *4.TU.ResearchData*, *Kaggl*, *DATA.GOV*, *DATAHUB* and *UCI Machine Learning Repository*.

The dataset I selected can be found at the following link:

‘https://archive.ics.uci.edu/ml/datasets/Activities+of+Daily+Living+%28ADLs%29+Recognition+Using+Binary
.

The dataset includes real information on the **daily life activities** performed by two users in their their homes. This dataset is composed by two instances of data, each one corresponding to a different user (which we call A and B). Each instance of the dataset is described by text files, i.e.: description, sensor events (characteristics), activities of daily living (labels).

4 Used technologies and tools

To carry out the following process mining work I decided to use *PM4Py*, an open source Python library that has a relatively wide range of functionality concerning the scope of the case study. Commercial software usually supports only one discovery algorithm and is very limited in terms of conformance checking, with the main emphasis on model visualization.

In contrast, *PM4Py*:

- supports basic algorithms for Process Discovery as *Alpha miner*, *Inductive miner* and *Heuristic miner*
- supports algorithms for Process Conformance
- contains algorithms and techniques for filtering logs
- allow detailed analysis of models and logs
- allows detailed diagnostics of the results

The development environment I decided to adopt, to relate the necessary python code, is **Jupyter Notebook**. The reason is because Jupyter allows one to create and share interactive textual documents containing executable code. Therefore, thanks to this aspect of it, it is possible for me to provide the report and the code in a single file that can be consulted easily and quickly.

5 Pre-implementation phase

The pre-implementation phase, first required downloading the *Anaconda platform*, an open-source suite containing several packages, libraries and applications specifically for Python. Among the various programs it also offers *Jupyter Notebook*.

To be able to use the *PM4Py library*, however, it was necessary from the *Anaconda Prompt* to create one's own environment, activate it and download python version 11 (the most recent). Still from the *Anaconda Prompt*, it was necessary to download *GraphViz*, a visualisation library used by *PM4Py*. Finally, the last step, before starting work, was to install *PM4Py* from the *Anaconda Prompt*.

6 Implementation phase

Before being able to work with the dataset, the *txt* files had to be converted into *csv* files to be used by the PM4Py functions. The conversion was done by simply uploading the files to *Microsoft Exel* and saving them in the desired format.

I then looked at the data to see if it was appropriate and I found that while there were activities and timestamps, there wasn't a data column that could serve as an identifier for the cases/traces.

Having learned this, based on the concept of process, I assumed that in the case of an individual's daily activities, the process consists of when one wakes up to when one goes to sleep. Thus I manually added a different numeric id for each group of activities ranging from when one go to sleep to the next time it is done.

6.1 Load packages

[1]:
```
import sys
import os            # creating and removing a directory (folder), fetching its␣
 ↪contents
import pandas as pd  # data processing, CSV file I/O (e.g. pd.read_csv)
import pm4py         # process mining and visualization
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
warnings.filterwarnings("ignore", category=UserWarning)
```

6.1.1 Environment

[2]:
```
pd.DataFrame([['platform', sys.platform], ['environment', sys.
 ↪prefix],['python', sys.version_info],], columns = ['System info', 'version']
).set_index('System info')
```

[2]:
```
                                           version
System info
```

```
platform                                             win32
environment  C:\Users\Giuseppe\anaconda3\envs\myenv
python                                 (3, 11, 0, final, 0)
```

6.1.2 Version

[3]:
```
pd.DataFrame([['pandas', pd.__version__], ['pm4py', pm4py.__version__],],␣
↪columns = ['package', 'version']).set_index('package')
```

[3]:
```
        version
package
pandas    1.5.3
pm4py     2.5.2
```

6.2 Load logs

[4]:
```
# reads the XES files of person A
sensors_log    = pm4py.read_xes('Dataset/OrdonezA_Sensors.xes')    #taking the␣
↪set of sensor actions
activities_log = pm4py.read_xes('Dataset/OrdonezA_Activities.xes') #taking the␣
↪set of daily human actions
all_events_log = pm4py.read_xes('Dataset/OrdonezA_All.xes')        #taking the␣
↪union set of both

activities_log_B = pm4py.read_xes('Dataset/OrdonezA_All.xes')      #taking the␣
↪set of daily human actions of person B

# Convert 'LifeCycles' to obj (this is done because LifeCycles to be used as␣
↪case_id must be a String)
sensors_log['LifeCycles']   =sensors_log['LifeCycles'].astype('object')
activities_log['LifeCycles']=activities_log['LifeCycles'].astype('object')
all_events_log['LifeCycles']=all_events_log['LifeCycles'].astype('object')
activities_log_B['LifeCycles']=activities_log_B['LifeCycles'].astype('object')
```

```
parsing log, completed traces ::   0%|          | 0/14 [00:00<?, ?it/s]

parsing log, completed traces ::   0%|          | 0/14 [00:00<?, ?it/s]

parsing log, completed traces ::   0%|          | 0/14 [00:00<?, ?it/s]

parsing log, completed traces ::   0%|          | 0/14 [00:00<?, ?it/s]
```

Data can be imported directly in *XES* or in *CSV* format using the *Pandas* library. Since *XES* is a widely used standard to represent data in this area, and since *PM4Py* suggests the use of this type of file, I decided to implement some python code capable of converting the files (already converted from txt to *CSV*) in *XES* files.

This code has been provided separately within the project folder as: 'CSV to XES format converter.ipynb'

6.3 Displays general information on the data set in order to choose the models to derive

[5]:
```
pd.set_option('display.max_rows', 4) #max number of lines that can be displayed
```

[6]:
```
#displays activities_logs
activities_log.iloc[:, [0, 1, 2, 7]]#useful column
```

[6]:
```
                         Start                       End       Activity  \
0   2011-11-28 02:27:00+00:00 2011-11-28 10:18:00+00:00       Sleeping
1   2011-11-28 10:21:00+00:00 2011-11-28 10:23:00+00:00      Toileting
..                         …                         …              …
246 2011-12-07 00:08:00+00:00 2011-12-07 00:54:00+00:00  Spare_Time/TV
247 2011-12-07 00:57:00+00:00 2011-12-07 00:57:00+00:00      Toileting

     @@case_index
0               0
1               0
..            …
246            13
247            13

[248 rows x 4 columns]
```

Looking at this first log, each human activity event is a line specifying the duration of the activity.

The following templates can be obtained from this log:

- **(1.1)** The model which shows the daily activities of person A over the entire time frame under consideration. Considering the *'activity'* column as activity, the *'Start'* column as timestamp and the *'LifeCycles'* column as the case_id.
- **(1.2)** The model describing the daily process of person A divided by days of the week. In this case it is necessary to introduce a column in the log associating each event with the corresponding day of the week.

[69]:
```
#displays sensors logs
sensors_log.iloc[:, [0, 1, 2, 3, 4,5]]#useful column
```

[69]:
```
                         Start                       End Location      Type  \
0   2011-11-28 02:27:00+00:00 2011-11-28 10:18:00+00:00      Bed  Pressure
1   2011-11-28 10:21:00+00:00 2011-11-28 10:21:00+00:00  Cabinet  Magnetic
2   2011-11-28 10:21:00+00:00 2011-11-28 10:23:00+00:00    Basin       PIR
3   2011-11-28 10:23:00+00:00 2011-11-28 10:23:00+00:00   Toilet     Flush
4   2011-11-28 10:25:00+00:00 2011-11-28 10:32:00+00:00   Shower       PIR
..                         …                         …      …        …
403 2011-12-06 19:25:00+00:00 2011-12-06 19:26:00+00:00    Basin       PIR
404 2011-12-06 19:40:00+00:00 2011-12-07 00:07:00+00:00     Seat  Pressure
405 2011-12-07 00:07:00+00:00 2011-12-07 00:07:00+00:00    Basin       PIR
406 2011-12-07 00:08:00+00:00 2011-12-07 00:54:00+00:00     Seat  Pressure
```

```
407 2011-12-07 00:57:00+00:00 2011-12-07 00:57:00+00:00    Toilet     Flush

        Place LifeCycles
0     Bedroom          1
1    Bathroom          1
2    Bathroom          1
3    Bathroom          1
4    Bathroom          1
..        …          …
403  Bathroom          9
404    Living          9
405  Bathroom          9
406    Living          9
407  Bathroom          9

[408 rows x 6 columns]
```

In this second log, one can see that each sensor event is a line specifying the location, the activation time (given by the time between two dates) and the room in which the sensor event is detected.

From this log we can derive the following models:

- **(2.1)** The model describing the order of execution of the sensors with respect to the entire daily life cycle. This is done by considering the *'Location'* column as the activity, the *'Start'* column as the timestamp and the *'LyfeCycles'* column as the case_id.
- **(2.2)** The model describing the order of execution of the sensors with respect to the individual rooms in which they are located. In this case the case_id will be the *'Place'* column.
- **(2.3)** The model describing the path executed by person A. Taking the *'Place'* column as the activity and the *'LifeCycles'* column as the case_id. So in this way we can reconstruct the path taken by the person within his or her apartment.

Before creating the models, we can observe the data through various other functions and analyses:

[8]:
```python
# Investigate what activities we have within the logs, including their␣
↪frequencies and considering all cases.
from pm4py.algo.filtering.log.attributes import attributes_filter
activities = attributes_filter.get_attribute_values(activities_log, "Activity")
sensors    = attributes_filter.get_attribute_values(sensors_log, "Location")
```

[9]:
```python
#list of daily human activitie and their occurrence
pd.set_option('display.max_rows', len(activities_log.value_counts('Activity')))␣
↪#max number of lines that can be displayed
pd.DataFrame(activities_log.value_counts('Activity'),columns = ['Count'])
```

[9]:
```
               Count
Activity
Spare_Time/TV     77
Grooming          51
```

```
Toileting         44
Breakfast         14
Leaving           14
Showering         14
Sleeping          14
Snack             11
Lunch              9
```

[10]:

```
#list of sensors:
pd.set_option('display.max_rows', len(sensors_log.value_counts('Location')))␣
↪#max number of lines that can be displayed
pd.DataFrame(sensors_log.value_counts('Location'),columns = ['Count'])
```

[10]:

	Count
Location	
Seat	81
Basin	70
Fridge	56
Toilet	45
Cupboard	34
Maindoor	31
Microwave	20
Cabinet	16
Bed	14
Shower	14
Toaster	14
Cooktop	13

[11]:

```
pd.set_option('display.max_rows', 4) #max number of lines that can be displayed
```

[12]:

```
start_sensor     = pm4py.get_start_activities(sensors_log)
end_sensor       = pm4py.get_end_activities  (sensors_log)
start_activities = pm4py.get_start_activities(activities_log)
end_activities   = pm4py.get_end_activities  (activities_log)

print("Start sensor: {}\nEnd sensor:   {}".format(start_sensor,end_sensor))
print("\nStart activities: {}\nEnd activities:   {}".
↪format(start_activities,end_activities))
```

```
Start sensor: {'Bed': 14}
End sensor:   {'Seat': 8, 'Basin': 5, 'Toilet': 1}

Start activities: {'Sleeping': 14}
End activities:   {'Spare_Time/TV': 8, 'Grooming': 4, 'Toileting': 2}
```

It is clear from the above data that the first activity of the daily cycle is (by choice) the time when it is recorded that person A goes to sleep. Furthermore, it can be observed that the last activity (before ending the cycle by going back to sleep) is always one of the following: spending free time (e.g. watching TV), taking care of his/her personal care/hygiene and using the toilet.

It is also noted that the initial and final recorded activities coincide perfectly with the initial and final sensors.

Therefore, even if present on two separate datasets, the two types of data are closely related (in fact, it is thanks to the sensor data, together with other elements, that it was possible to recognize the activities of individual A.

Indeed the publisher of the datasets invites the inclusion of both in the same experimental set-up due to their similarities.

For this reason, I created a single dataset containing both data (named: *OrdonezA_All*), matching the *'start'* and *'end'* columns and joining the *'location'* and *'activities'* column into one.

Shown below:

[13]:
```
#displays some lines of all_logs
all_events_log.iloc[:, [0, 1, 2, 3, 4,5]]#useful column
```

[13]:
```
                          Start                       End   Location      Type  \
0   2011-11-28 02:27:00+00:00 2011-11-28 10:18:00+00:00        Bed  Pressure
1   2011-11-28 02:27:00+00:00 2011-11-28 10:18:00+00:00   Sleeping       NaN
..                          …                         …          …        …
654 2011-12-07 00:57:00+00:00 2011-12-07 00:57:00+00:00     Toilet     Flush
655 2011-12-07 00:57:00+00:00 2011-12-07 00:57:00+00:00  Toileting       NaN

        Place LifeCycles
0     Bedroom          1
1         NaN          1
..        …          …
654  Bathroom          9
655       NaN          9

[656 rows x 6 columns]
```

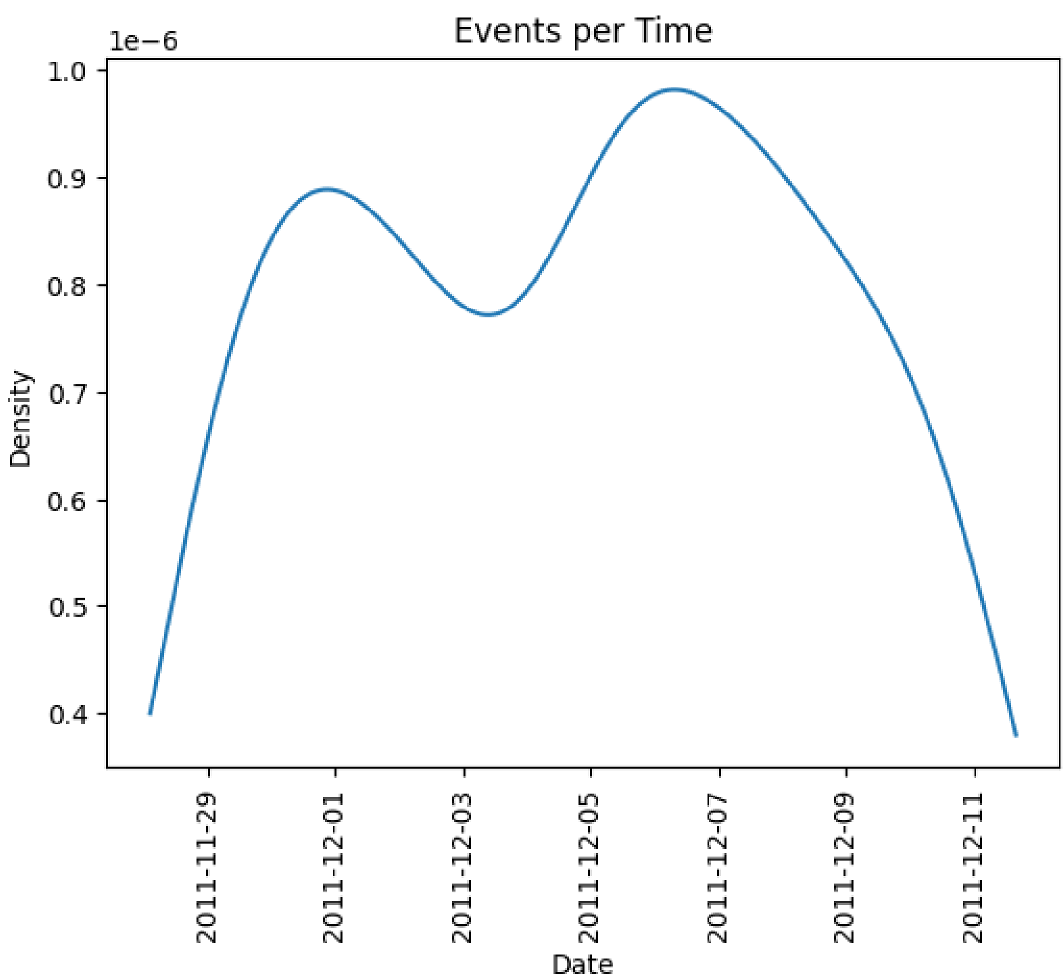
From this third log, by combining activities and sensors in the *'Location'* column, we can derive:

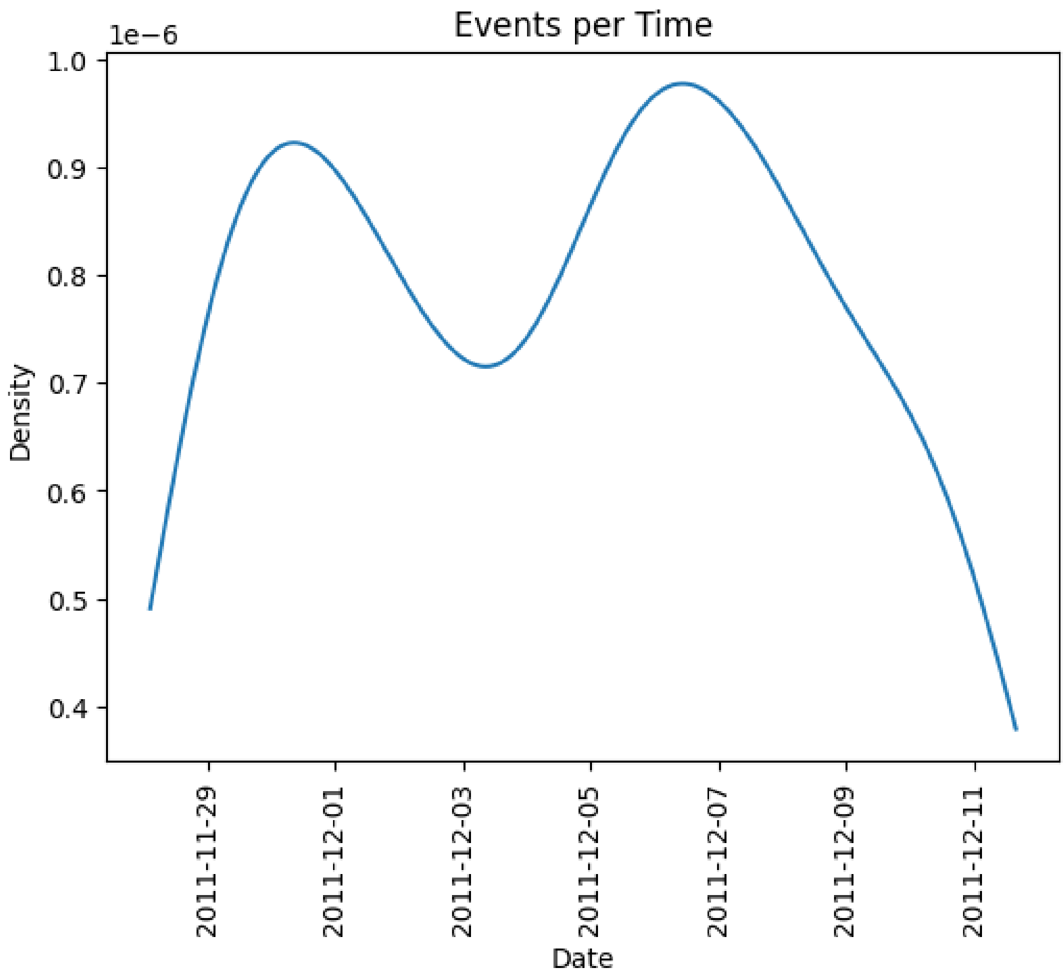
- **3.1)** the model of daily human activities that includes the appliances or sensors with which the person interacted.

6.3.1 Distribution of activities agains different time scales

[14]:
```
# Distribution of events over time
# It helps to understand in which time intervals the greatest number of events␣
 ↪is recorded.
from pm4py.algo.filtering.log.attributes import attributes_filter

x, y   = attributes_filter.get_kde_date_attribute(activities_log,␣
 ↪attribute="time:timestamp")
```

```
x2, y2 = attributes_filter.get_kde_date_attribute(sensors_log,
    attribute="time:timestamp")

# Visualize it
from pm4py.visualization.graphs import visualizer as graphs_visualizer

gviz  = graphs_visualizer.apply_plot(x, y,   variant=graphs_visualizer.Variants.
    DATES)
gviz2 = graphs_visualizer.apply_plot(x2, y2, variant=graphs_visualizer.Variants.
    DATES)
graphs_visualizer.view(gviz)
graphs_visualizer.view(gviz2)
```



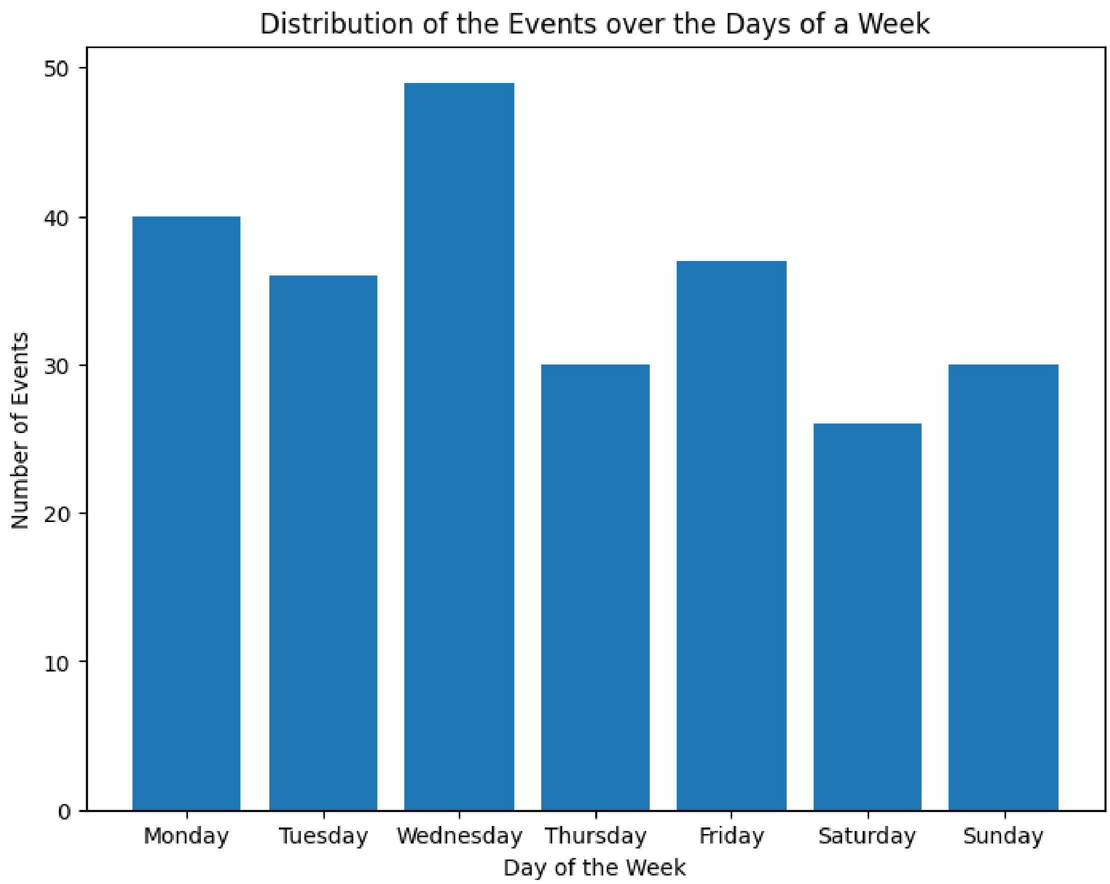


These graphs show the distribution of activities and events recorded by the sensors during the entire data collection period, which took place between 28 November and 11 December 2011 for a total period of 13 days or nearly two weeks.

It can be seen that the two graphs have the same distribution curve which confirms the relationship between these data types. Both show an initial growth of events up to 1 December, then a subsequent drop in events up to 3 December, a new growth leading to the highest peak of events around 7 December and finally a further drop up to 11 December .

To understand the reason for these variations, we should consider the days of the week:

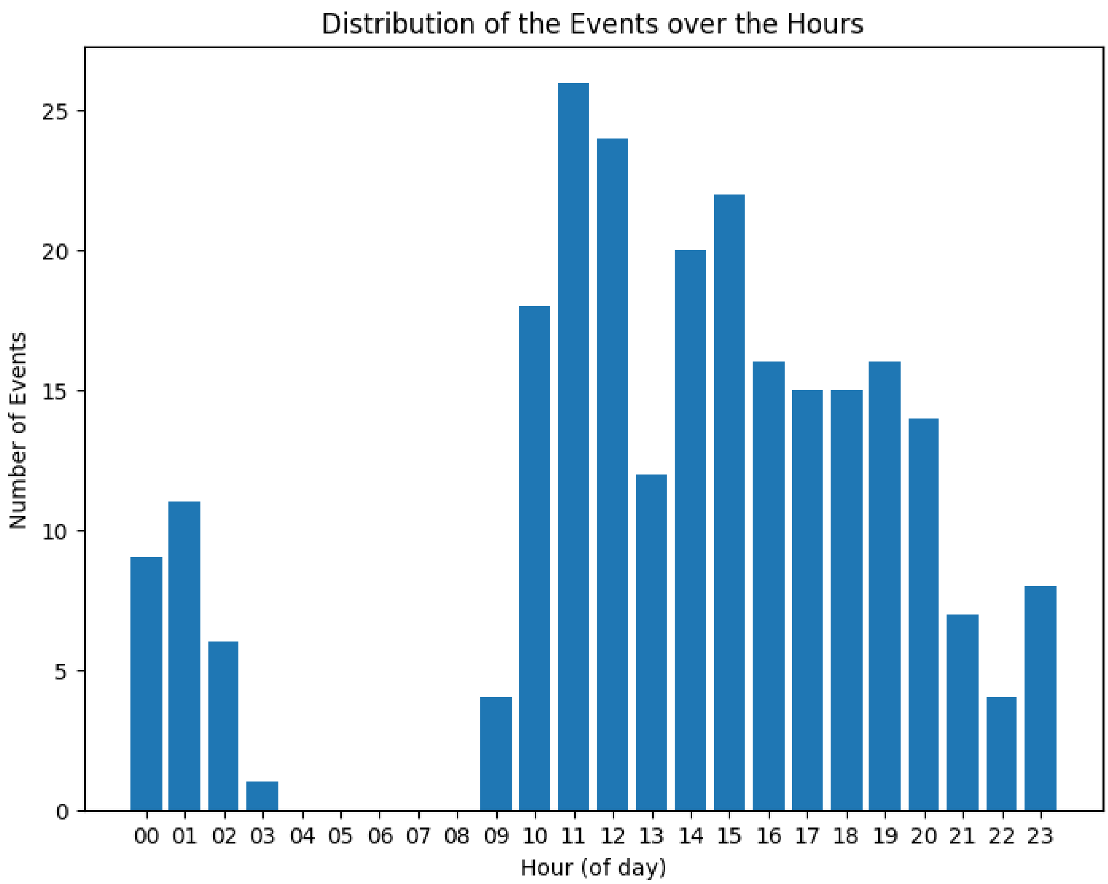
[15]:
```
# Distribution of activities agains different time scales
pm4py.view_events_distribution_graph(activities_log, distr_type="days_week", ␣
↪format="png")
```



We can observe that the most intense day (where most activities are recorded) is Wednesday followed by Monday and Friday while the least intense is Saturday followed by Sunday and Thursday

We can also observe the distribution in the different daily times:

[16]:
```
pm4py.view_events_distribution_graph(activities_log, distr_type="hours", 
↪format="png")
```



From this graph it can be seen that the person is not involved in activities from 4 a.m. to 8 a.m., so he or she sleeps deeply until 9 e.m. and certainly goes to sleep before 3 a.m. Furthermore, the busiest period of the day is recorded between 10 and 12 a.m. and then between 2 and 8 p.m.

Variants:

[17]:
```
vars = pm4py.get_variants(activities_log)
for k, v in vars.items():
    print('{}'.format('--> '.join(k)), " ",v, " \n")
print("Number of variants: ",len(vars))
```

```
Sleeping--> Toileting--> Showering--> Breakfast--> Grooming--> Spare_Time/TV-->
Toileting--> Leaving--> Spare_Time/TV--> Toileting--> Lunch--> Grooming-->
Spare_Time/TV--> Snack--> Spare_Time/TV   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Spare_Time/TV-->
Lunch--> Spare_Time/TV--> Grooming--> Toileting--> Spare_Time/TV--> Grooming-->
Leaving--> Grooming--> Toileting--> Spare_Time/TV--> Snack--> Snack-->
Toileting--> Spare_Time/TV--> Spare_Time/TV--> Toileting--> Spare_Time/TV   1

Sleeping--> Grooming--> Toileting--> Grooming--> Showering--> Breakfast-->
```

```
Spare_Time/TV--> Grooming--> Leaving--> Spare_Time/TV--> Grooming-->
Spare_Time/TV--> Grooming--> Spare_Time/TV--> Grooming   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Spare_Time/TV-->
Grooming--> Leaving--> Spare_Time/TV--> Toileting--> Spare_Time/TV-->
Grooming--> Spare_Time/TV--> Leaving--> Spare_Time/TV--> Grooming-->
Spare_Time/TV--> Grooming--> Spare_Time/TV   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Spare_Time/TV-->
Grooming--> Leaving--> Spare_Time/TV--> Grooming--> Leaving--> Spare_Time/TV-->
Grooming   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Spare_Time/TV-->
Lunch--> Toileting--> Grooming--> Spare_Time/TV   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Grooming-->
Spare_Time/TV--> Snack--> Spare_Time/TV--> Toileting--> Lunch--> Grooming-->
Spare_Time/TV--> Toileting--> Spare_Time/TV--> Toileting--> Spare_Time/TV-->
Leaving--> Spare_Time/TV   1

Sleeping--> Toileting--> Showering--> Breakfast--> Grooming--> Spare_Time/TV-->
Snack--> Spare_Time/TV--> Toileting--> Spare_Time/TV--> Lunch-->
Spare_Time/TV--> Toileting--> Snack--> Spare_Time/TV--> Toileting-->
Spare_Time/TV--> Toileting--> Spare_Time/TV--> Toileting--> Leaving-->
Grooming--> Spare_Time/TV   1

Sleeping--> Grooming--> Showering--> Breakfast--> Spare_Time/TV--> Lunch-->
Spare_Time/TV--> Toileting--> Spare_Time/TV--> Grooming--> Leaving-->
Spare_Time/TV--> Toileting--> Spare_Time/TV--> Grooming--> Spare_Time/TV-->
Grooming--> Spare_Time/TV--> Grooming   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Spare_Time/TV-->
Lunch--> Spare_Time/TV--> Grooming--> Spare_Time/TV--> Toileting-->
Spare_Time/TV--> Snack--> Spare_Time/TV--> Toileting--> Spare_Time/TV   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Spare_Time/TV-->
Grooming--> Spare_Time/TV--> Grooming--> Leaving--> Spare_Time/TV-->
Toileting--> Spare_Time/TV   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Spare_Time/TV-->
Toileting--> Leaving--> Spare_Time/TV--> Snack--> Spare_Time/TV--> Grooming-->
Spare_Time/TV--> Toileting--> Snack--> Spare_Time/TV--> Toileting-->
Spare_Time/TV--> Grooming   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Spare_Time/TV-->
Grooming--> Leaving--> Lunch--> Grooming--> Toileting--> Spare_Time/TV-->
Toileting--> Spare_Time/TV--> Leaving--> Spare_Time/TV--> Snack-->
Spare_Time/TV--> Grooming--> Toileting--> Spare_Time/TV--> Toileting-->
```

```
Grooming--> Spare_Time/TV--> Toileting   1

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Spare_Time/TV-->
Lunch--> Grooming--> Spare_Time/TV--> Toileting--> Spare_Time/TV--> Grooming-->
Spare_Time/TV--> Snack--> Grooming--> Spare_Time/TV--> Grooming-->
Spare_Time/TV--> Toileting   1

Number of variants:  14
```

With *PM4Py* it is possible to obtain a list of all **variants** and their occurrence. In our case there are 14 variants each occurring in one trace. A variant represents a unique sequence of transitions (in our case activities).

Common activities:

[67]:
```
pd.set_option('display.max_rows', 50)
# Create a table giving the number of activities in which each activity is␣
↪present.
number_activities = pd.crosstab(activities_log['LifeCycles'],␣
↪activities_log['Activity'])
```

[66]:
```
## Calculate the number of unique activities counts
## This should be 1 for activities which are shared by all LifeCycles.
n_unique = number_activities.apply(pd.Series.nunique)
## Identify the events which are shared by all
shared_activities  = n_unique[n_unique==1].index
activities_to_keep = n_unique[n_unique >1].index
print('The following activities are common to all cases: \n -{}'.format(' \n -'.
↪join(shared_activities)))
print('The following activities are the ones that we wish to keep (not common␣
↪to all cases): \n -{}'.format(' \n -'.join(activities_to_keep)))
```

```
The following activities are common to all cases:
 -Breakfast
 -Showering
 -Sleeping
The following activities are the ones that we wish to keep (not common to all
cases):
 -Grooming
 -Leaving
 -Lunch
 -Snack
 -Spare_Time/TV
 -Toileting
```

6.4 (1) Process Discovery on the activities_log

With Process discovery we aim to find a suitable process model that can describe our process and the sequence of events (traces) and activities that are performed within each trace.

There are several algorithms that can be applied with *PM4Py* to obtain a model (each with its own advantages and disadvantages)

One by one, below, I derived the patterns identified in the log analysis phase.

6.4.1 (1.1) The model of daily activities (of person A) over the entire time interval considered.

6.4.2 Alpha Miner

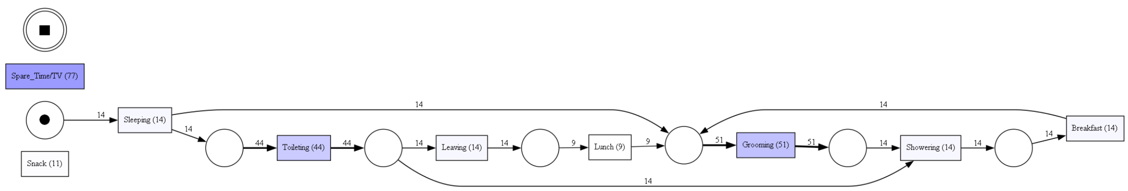
Alpha miner is one of the most best-known process mining algorithms, and one of the first algorithm invented for process discovery. The Alpha Miner algorithm is particularly well-suited for discovering processes in which concurrent activities are common, such as in the case of distributed systems or processes involving multiple stakeholders.

The output of the Alpha Miner is an Petri Net.

- Advantages
  - Creates simple models that agree with the quality criteria of Generalization and Simplicity
- Disadvantages
  - It lacks Precision and Fitness as a consequence of the quality criteria it possesses.
  - does not take into account event frequencies.

[20]:
```
# Apply Halpa miner algorithm
from pm4py.algo.discovery.alpha import algorithm as alpha_miner        # Import␣
↪the Halpa miner algorithm
from pm4py.visualization.petri_net import visualizer as pn_visualizer  # Import␣
↪the petri-net visualization object

net, initial_marking, final_marking = alpha_miner.apply(activities_log)

parameters = {pn_visualizer.Variants.FREQUENCY.value.Parameters.FORMAT: "png"}

gviz = pn_visualizer.apply(net, initial_marking, final_marking,␣
↪parameters=parameters,
                           variant=pn_visualizer.Variants.FREQUENCY,␣
↪log=activities_log)

pn_visualizer.view(gviz)
```

```
replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
↪it/s]
```



The generated model is really very simple. First thing you notice is that the transitions 'snack' and 'spare time' have not been linked to any place. This is because probably if connected, loops of length one and length two (not allowed by the algorithm) would be formed.

The Petri Net model obtained:

- is not *Safeness* since the place between *'Grooming'* and *'showering'* is 2-bounded.
- is not *Safeness* since the place between *'Grooming'* and *'showering'* is 2-bounded.
- is not *deadlock-free* since if when you return for the second time to the place between 'Grooming' and 'showering' there is no possibility of firing the only transition available.

It is also possible to automatically verify that the obtained model is soudness.

A Petri Net is *sound* iff:

- It is well formed.
- it contains no live-locks.
- it contains no deadlocks.
- we are able to always reach the final marking.

[21]:
```
from pm4py.algo.analysis.woflan import algorithm as woflan
#check_soundness
woflan.apply(net, initial_marking, final_marking)
```

```
Input is ok.
There is more than one source place.
```

[21]:
```
False
```

By constructing the reachability graph I was able to find the possible sequences of actions that the model allows:

Sleeping –> Grooming –> Toileting –> Showering –> Lunch –> Grooming

Sleeping –> Grooming –> Toileting –> Leaving –> Lunch –> Grooming

Sleeping –> Toileting –> Grooming –> Showering –> Breakfast –> Grooming

Sleeping –> Toileting –> Grooming –> Leaving –> Lunch –> Grooming

Sleeping –> Toileting –> Leaving –> Grooming

Number of variants: 5

It is evident that the possible sequences are not exhaustive even if the actions of *'Sneck'* and *'Spare_Time/TV'* are eliminated as demonstrated below:

[22]:
```
filtered = pm4py.filter_event_attribute_values(activities_log,'concept:
↪name',{'Snack','Spare_Time/TV'}, level='event', retain=False)
vars = pm4py.get_variants(filtered)
for k, v in vars.items():
    print('{}'.format('--> '.join(k)), " \n")
print("Number of variants: ",len(vars))
```

```
Sleeping--> Toileting--> Showering--> Breakfast--> Grooming--> Toileting-->
Leaving--> Toileting--> Lunch--> Grooming

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Lunch-->
Grooming--> Toileting--> Grooming--> Leaving--> Grooming--> Toileting-->
Toileting--> Toileting

Sleeping--> Grooming--> Toileting--> Grooming--> Showering--> Breakfast-->
Grooming--> Leaving--> Grooming--> Grooming--> Grooming

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Grooming-->
Leaving--> Toileting--> Grooming--> Leaving--> Grooming--> Grooming

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Grooming-->
Leaving--> Grooming--> Leaving--> Grooming

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Lunch-->
Toileting--> Grooming

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Grooming-->
Toileting--> Lunch--> Grooming--> Toileting--> Toileting--> Leaving

Sleeping--> Toileting--> Showering--> Breakfast--> Grooming--> Toileting-->
Lunch--> Toileting--> Toileting--> Toileting--> Toileting--> Leaving--> Grooming

Sleeping--> Grooming--> Showering--> Breakfast--> Lunch--> Toileting-->
Grooming--> Leaving--> Toileting--> Grooming--> Grooming--> Grooming

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Lunch-->
Grooming--> Toileting--> Toileting

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Grooming-->
Grooming--> Leaving--> Toileting

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Toileting-->
Leaving--> Grooming--> Toileting--> Toileting--> Grooming

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Grooming-->
```

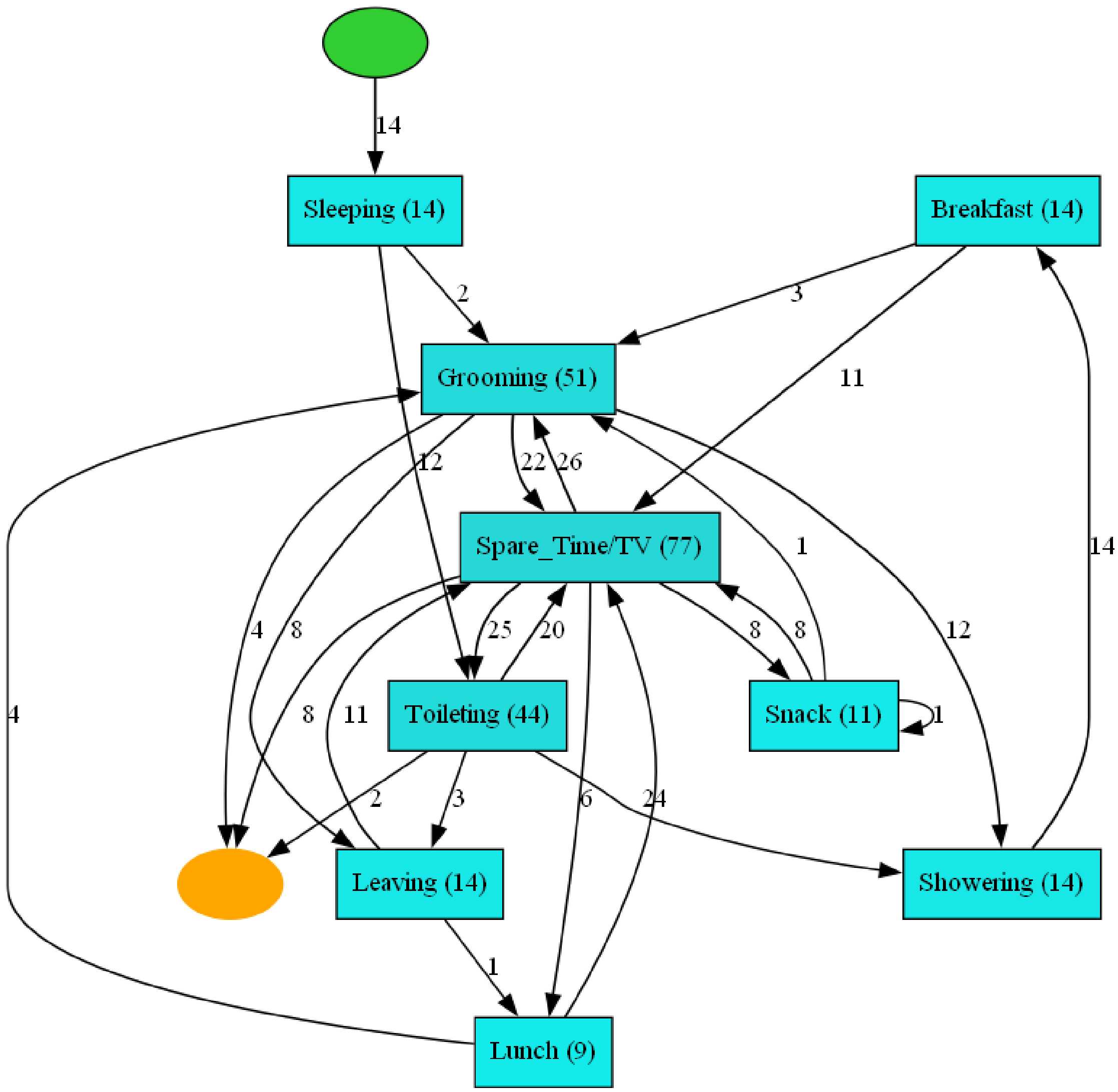
```
Leaving--> Lunch--> Grooming--> Toileting--> Toileting--> Leaving--> Grooming-->
Toileting--> Toileting--> Grooming--> Toileting

Sleeping--> Toileting--> Grooming--> Showering--> Breakfast--> Lunch-->
Grooming--> Toileting--> Grooming--> Grooming--> Grooming--> Toileting

Number of variants:  14
```

6.4.3 Heuristic Miner

Heuristics miner takes into account event frequencies and ignores exceptional behaviour (low frequency events and event sequences), single events, and short loops. The output of the Heuristics Miner is an Heuristics Net represented as a flowchart/Process map.

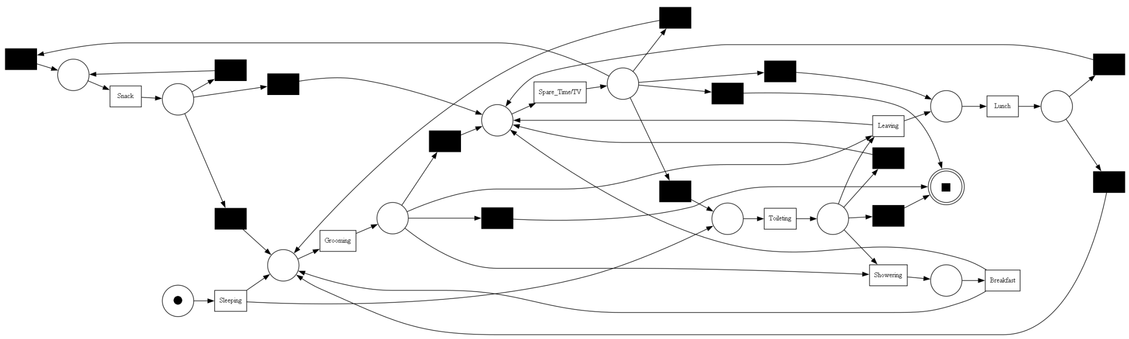
- Advantages
  - applies filtering to reduce noise
  - detects short loops
- Disadvantages
  - It lacks Precision and Fitness as a consequence of the quality criteria it possesses.

[23]:
```
# Apply Heuristic miner algorithm
heu_net = pm4py.discover_heuristics_net(activities_log,
                                    dependency_threshold=0.5, and_threshold=0.
 ↪65, loop_two_threshold=0.7,
                                    min_act_count=1, min_dfg_occurrences=1)
pm4py.view_heuristics_net(heu_net)
```



This process map looks much cleaner, it contains only activities and connections. Each connection represents a precedence relation. Consequentially concurrent activities are mutually connected by arcs in both directions.

[24]:
```
#Converting the Heuristic Net into petri-net
petri_heuristics, im_heuristics, fm_heuristics = pm4py.
↪discover_petri_net_heuristics(activities_log,
                                 dependency_threshold=0.5, and_threshold=0.
↪65, loop_two_threshold=0.7)
pm4py.view_petri_net(petri_heuristics, im_heuristics, fm_heuristics,␣
↪format="png")
```

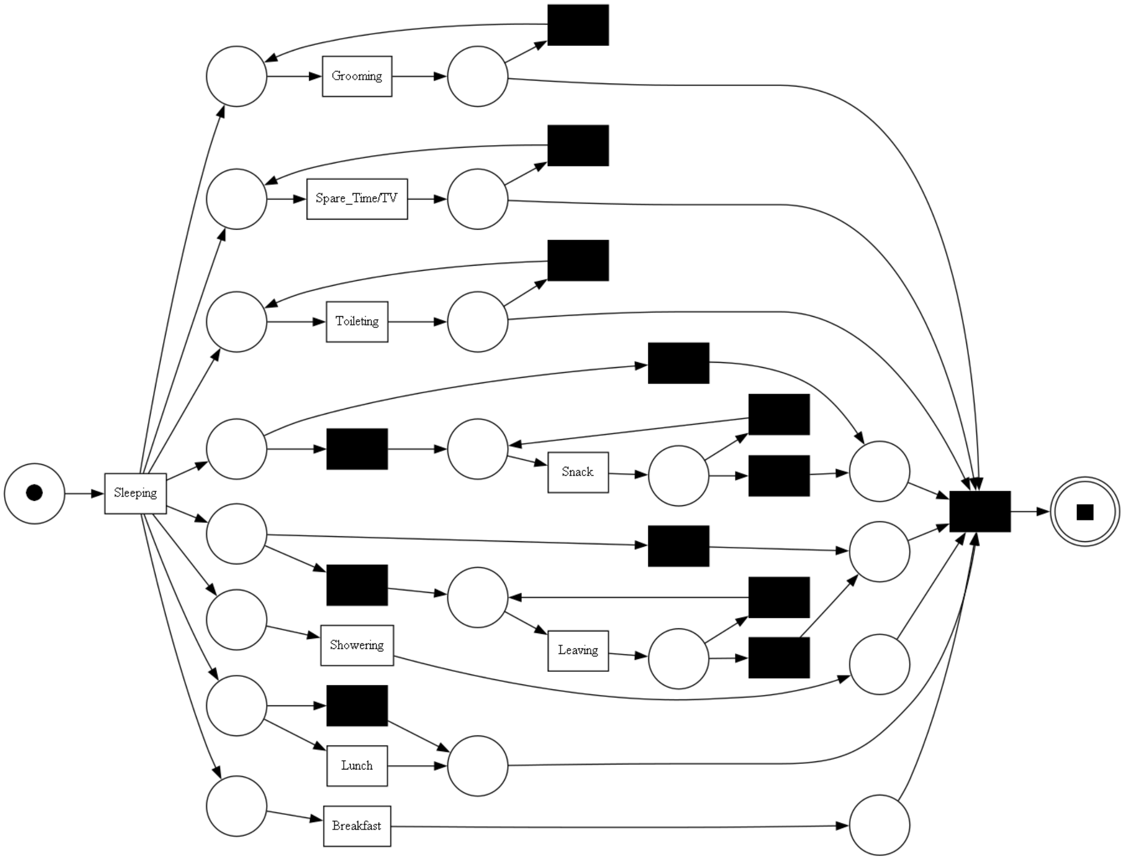


6.4.4 Inductive Miner

The Inductive Miner is the most usend process mining algorithm. Like heuristics miner, it takes into account event frequencies and ignores low-frequency events and isolated event loops. Two process models can be derived: Petri Net and Process Tree.

- Advantages
  - Guarantees Precision and Fitness
  - Can handle invisible task
- Disadvantages
  - It lacks Simplicity and Generalization.
  - Usually make extensive use of hidden transitions (especially for skipping/looping on a portion on the model).

[25]:
```
# Apply Inductive miner algorithm
petri_inductive, im_inductive, fm_inductive = pm4py.discover_petri_net_inductive(activities_log)
pm4py.view_petri_net(petri_inductive, im_inductive, fm_inductive, format="png")
```



From this Petri Net, compared to the one constructed by the Alph miner, it is possible to repeat each activity several times during the daily life cycle of person A, except for breakfast, showering and lunch which can only be done once. In addition, lunch, snack and leaving can even be skipped.

The Petri Net model obtained:

- is *Safeness* since every place is 1-bounded.
- is *deadlock-free* since does not contain any deadlock.

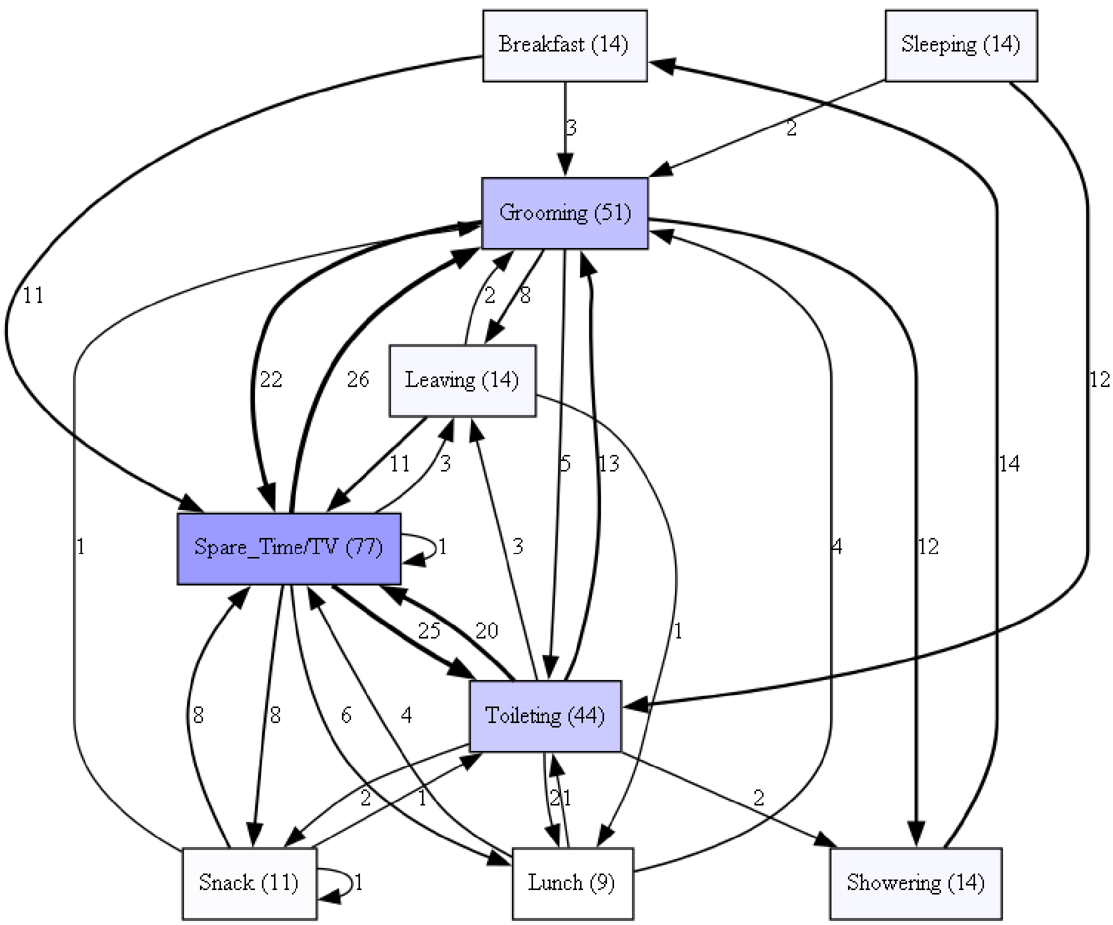
It is evident that the possible sequences are exhaustive, but there are so many of them that they risk allowing unrealistic situations as well.

6.4.5 DFG discovery

To better understand what the process models are capturing and omitting, it is worth looking at how the models discovered differ from the directly-follows graph of events and event transitions.

[26]:
```
# Apply dfg_discovery algorithm

from pm4py.algo.discovery.dfg import algorithm as dfg_discovery     # Import␣
↪the dfg_discovery algorithm
```

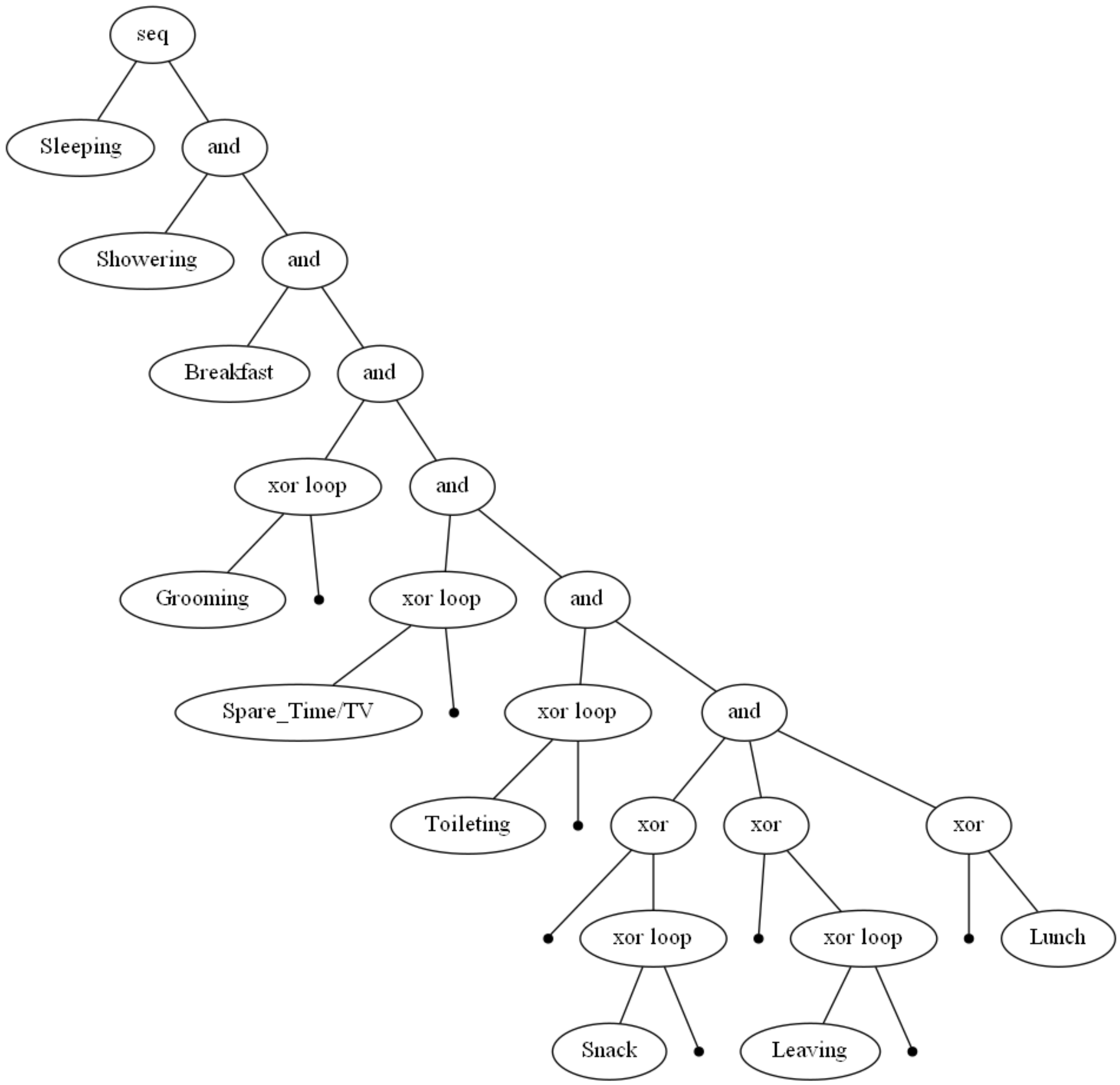
```python
from pm4py.visualization.dfg import visualizer as dfg_visualization # Import the dfg visualization object

#Create graph from log
dfg, dfg_sa, dfg_ea = pm4py.discover_dfg(activities_log)
# Visualise
gviz = dfg_visualization.apply(dfg, log=activities_log, variant=dfg_visualization.Variants.FREQUENCY)
dfg_visualization.view(gviz)
```



6.4.6 Tree Inductive

To better understand what the process models are capturing and omitting, it is worth looking at how the models discovered differ from the directly-follows graph of events and event transitions.

[27]:
```python
# Apply Tree inductive algorithm
tree_inductive = pm4py.discover_process_tree_inductive(activities_log)
pm4py.view_process_tree(tree_inductive, format="png")
```



The process tree is composed of 4 internal nodes:

- Sequence behavior: represents that the child nodes are a sequence of behaviors from left to right
- choice behavior: represents an exclusive choice, i.e. one of the children is executed
- cuncurrent behavior: represents the simultaneous behavior; therefore, his children can be executed simultaneously or in any order.
- looping behavior: represents the 'repeated behaviour', ie the possibility of repeating the behaviour.

Consequently this process tree shows that the first activity is 'sleeping', followed by any one of all the others (In unspecified order)

Where, however, 'spare time', 'toileting', 'grooming', 'snacks', 'leaving' and 'lunch' may not be performed. Furthermore, all activities except ‘breakfast’, ‘showering’, ‘sleeping’ and ‘lunch’ can be repeated several times.

In conclusion, the model is quite realistic as regards the occurrences of the activities but not very exhaustive as regards their temporal relationship. This model is quite realistic with regard to the occurrences of activities but not very comprehensive with regard to their temporal relationship.

[28]:
```
path="Models/1.1/"
#Save models as png
pm4py.save_vis_petri_net(net, initial_marking, final_marking, path+"alpha.png")
pm4py.save_vis_petri_net(petri_heuristics, im_heuristics, fm_heuristics,␣
 ↪path+"heuristics.png")
pm4py.save_vis_petri_net(petri_inductive, im_inductive, fm_inductive,␣
 ↪path+"inductive.png")
pm4py.save_vis_process_tree(tree_inductive, path+"inductive_tree.png")
pm4py.save_vis_heuristics_net(heu_net, path+"heunet.png")
pm4py.save_vis_dfg(dfg, dfg_sa, dfg_ea, path+"dfg.png")

#Save models as Dictionary
pm4py.write_pnml(net, initial_marking, final_marking, path+"Dictionary/alpha.
 ↪pnml")
pm4py.write_pnml(petri_heuristics, im_heuristics, fm_heuristics,␣
 ↪path+"Dictionary/heuristics.pnml")
pm4py.write_pnml(petri_inductive, im_inductive, fm_inductive, path+"Dictionary/
 ↪inductive.pnml")
pm4py.write_ptml(tree_inductive, path+"Dictionary/inductive.ptml")
pm4py.write_dfg(dfg, dfg_sa, dfg_ea, path+"Dictionary/dfg.dfg")
```

6.4.7 (1.2) The model of daily activities (of person A) on individual days of the week.

In order to derive the model, first of all I added a column in the log showing the day of the week:

[29]:
```
pd.set_option('display.max_rows', 7)
#Adding a column showing the current day of the week for each event date
activities_log['DayOfWeek']= pd.to_datetime(activities_log["time:timestamp"])
activities_log['DayOfWeek']= activities_log['DayOfWeek'].dt.day_name()
activities_log[["time:timestamp", 'DayOfWeek']]
```

[29]:
```
               time:timestamp  DayOfWeek
0   2011-11-28 02:27:00+00:00     Monday
1   2011-11-28 10:21:00+00:00     Monday
2   2011-11-28 10:25:00+00:00     Monday
..                        …          …
245 2011-12-07 00:07:00+00:00  Wednesday
246 2011-12-07 00:08:00+00:00  Wednesday
247 2011-12-07 00:57:00+00:00  Wednesday

[248 rows x 2 columns]
```

After that, I filtered the data by day of the week getting new logs each containing the specific events for that day:

[30]:
```
slice_Monday    = pm4py.
 ↪filter_event_attribute_values(activities_log,'DayOfWeek',{'Monday'},   ␣
 ↪level='event', retain=True)
slice_Tuesday   = pm4py.
 ↪filter_event_attribute_values(activities_log,'DayOfWeek',{'Tuesday'},  ␣
 ↪level='event', retain=True)
slice_Wednesday = pm4py.
 ↪filter_event_attribute_values(activities_log,'DayOfWeek',{'Wednesday'},␣
 ↪level='event', retain=True)
slice_Thursday  = pm4py.
 ↪filter_event_attribute_values(activities_log,'DayOfWeek',{'Thursday'}, ␣
 ↪level='event', retain=True)
slice_Friday    = pm4py.
 ↪filter_event_attribute_values(activities_log,'DayOfWeek',{'Friday'},   ␣
 ↪level='event', retain=True)
slice_Saturday  = pm4py.
 ↪filter_event_attribute_values(activities_log,'DayOfWeek',{'Saturday'}, ␣
 ↪level='event', retain=True)
slice_Sunday    = pm4py.
 ↪filter_event_attribute_values(activities_log,'DayOfWeek',{'Sunday'},   ␣
 ↪level='event', retain=True)

Monday    = pm4py.convert_to_event_log(slice_Monday)
Tuesday   = pm4py.convert_to_event_log(slice_Tuesday)
Wednesday = pm4py.convert_to_event_log(slice_Wednesday)
Thursday  = pm4py.convert_to_event_log(slice_Thursday)
Friday    = pm4py.convert_to_event_log(slice_Friday)
Saturday  = pm4py.convert_to_event_log(slice_Saturday)
Sunday    = pm4py.convert_to_event_log(slice_Sunday)
```

Then, I created a function to be called every time I want to generate the models and save them:

[31]:
```
def create_and_save_all_models(log, path):
    """
    create_and_save_all_models creates and saves models for a specified path␣
 ↪and log.

    :param log: vent log / Pandas dataframe (that you want to transform into a␣
 ↪model)
    :param path: path                       (where to save the images of the␣
 ↪generated model)
    """
    #create models
```

```python
    dfg, dfg_sa, dfg_ea = pm4py.discover_dfg(log)                # dfg_discovery   algorithm
    petri_alpha, im_alpha, fm_alpha = pm4py.discover_petri_net_alpha(log)                # Alpha inductive algorithm
    petri_inductive, im_inductive, fm_inductive = pm4py.discover_petri_net_inductive(log)      # Inductive miner algorithm
    petri_heuristics, im_heuristics, fm_heuristics = pm4py.discover_petri_net_heuristics(log) # Heuristic miner algorithm
    tree_inductive = pm4py.discover_process_tree_inductive(log)                # Tree inductive  algorithm
    heu_net = pm4py.discover_heuristics_net(log)                # Heuristic miner algorithm
    #
    #Save models as png
    pm4py.save_vis_dfg(dfg, dfg_sa, dfg_ea, path+"dfg.png")
    pm4py.save_vis_petri_net(petri_alpha, im_alpha, fm_alpha, path+"alpha.png")
    pm4py.save_vis_petri_net(petri_inductive, im_inductive, fm_inductive, path+"inductive.png")
    pm4py.save_vis_petri_net(petri_heuristics, im_heuristics, fm_heuristics, path+"heuristics.png")
    pm4py.save_vis_process_tree(tree_inductive, path+"inductive_tree.png")
    pm4py.save_vis_heuristics_net(heu_net, path+"heunet.png")
    #Save models as Dictionary
    pm4py.write_pnml(petri_alpha, im_alpha, fm_alpha, path+"Dictionary/alpha.pnml")
    pm4py.write_pnml(petri_heuristics, im_heuristics, fm_heuristics, path+"Dictionary/heuristics.pnml")
    pm4py.write_pnml(petri_inductive, im_inductive, fm_inductive, path+"Dictionary/inductive.pnml")
    pm4py.write_ptml(tree_inductive, path+"Dictionary/inductive.ptml")
    try:
        pm4py.write_dfg(dfg, dfg_sa, dfg_ea, path+"Dictionary/dfg.dfg")
    except:
        print("Dfg dictionary not saved")
```

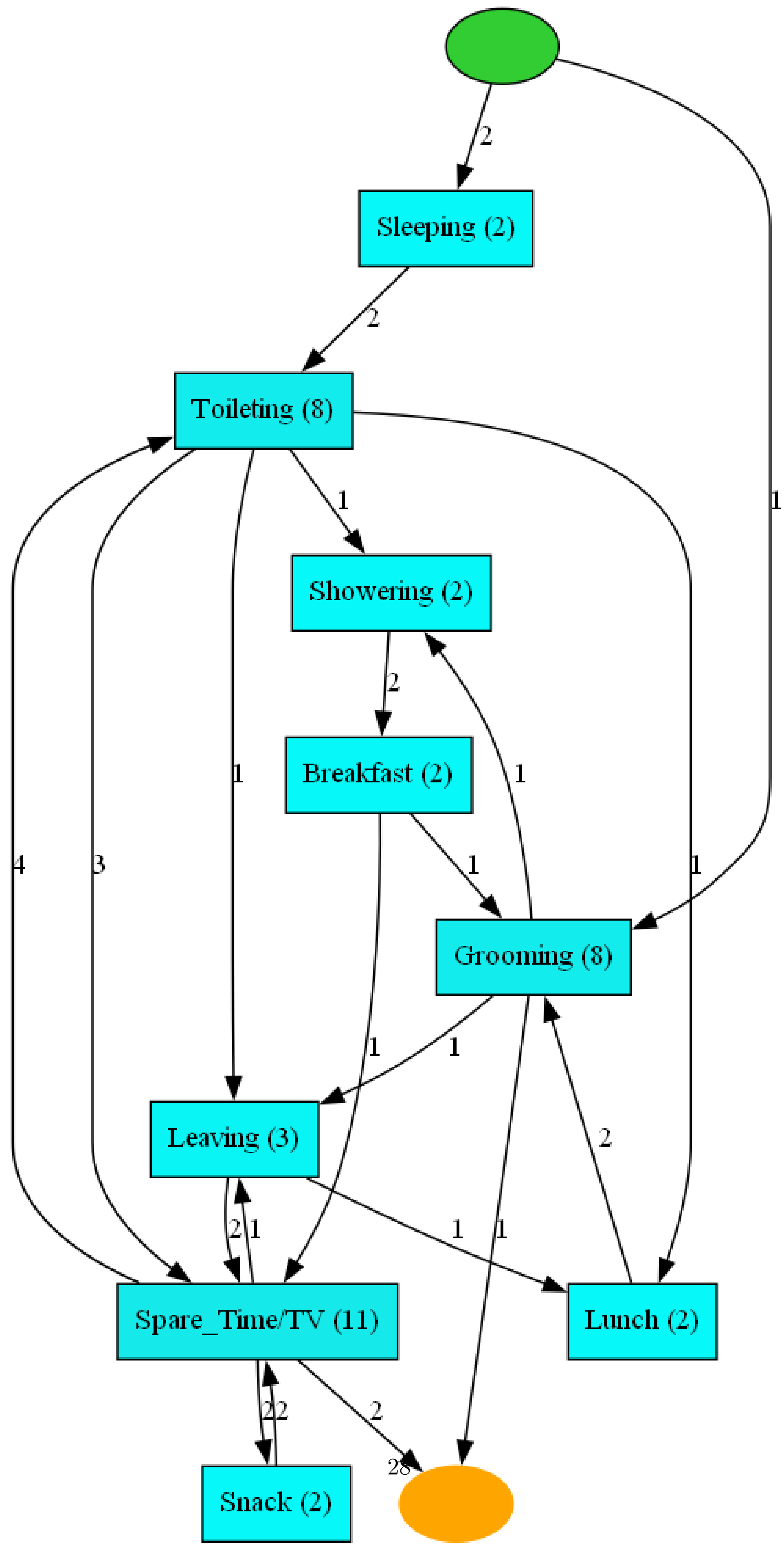
Finally, by using a for loop I derived the models for each log and saved them (in the folder: *'Models/1.2/'*).

[32]:
```python
def namestr(obj, namespace):
    return [name for name in namespace if namespace[name] is obj]

week_log = [Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday]

for i in week_log:
    #select Path
    folderName= namestr(i, globals())
    path="Models/1.2/"+folderName[0] + "/"
```

```python
    print("saving models in: ", path)
    #create_and_save function
    create_and_save_all_models(i,path)
```

```
saving models in:  Models/1.2/Monday/
saving models in:  Models/1.2/Tuesday/
saving models in:  Models/1.2/Wednesday/
saving models in:  Models/1.2/Thursday/
saving models in:  Models/1.2/Friday/
saving models in:  Models/1.2/Saturday/
saving models in:  Models/1.2/Sunday/
```

Let's take a look at one of the derived models as an example:



Sleeping (2)
Toileting (8)
Showering (2)
Breakfast (2)
Grooming (8)
Leaving (3)
Spare_Time/TV (11)
Lunch (2)
Snack (2)
28

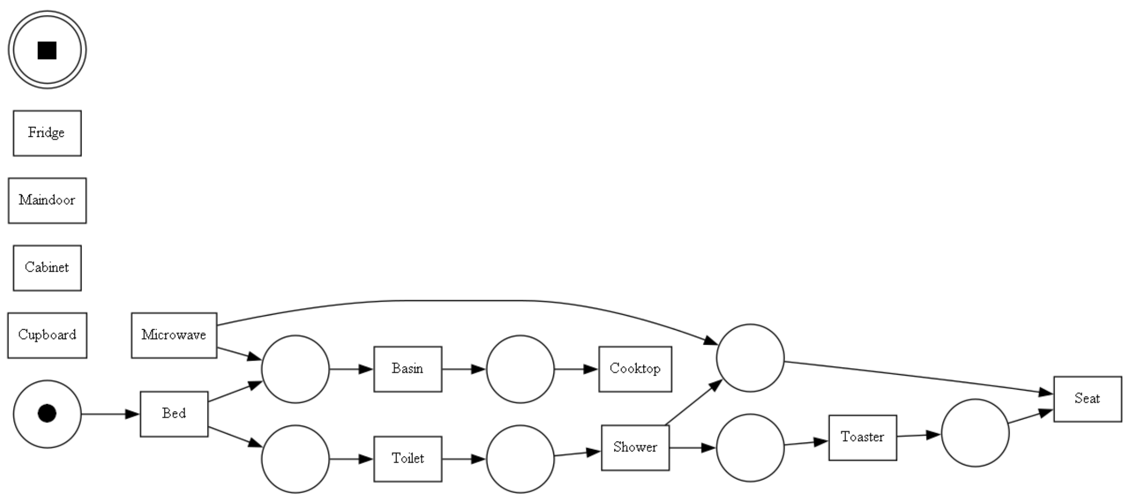
6.5 (2) Process Discovery on the sensors_log

6.5.1 (2.1) The sensor activiation model with respect to the entire time

Alpha Miner

[34]:
```
# Apply Alpha miner algorithm
net, initial_marking, final_marking = alpha_miner.apply(sensors_log)

parameters = {pn_visualizer.Variants.FREQUENCY.value.Parameters.FORMAT: "png"}

gviz = pn_visualizer.apply(net, initial_marking, final_marking,␣
 ↪parameters=parameters,
                            variant=pn_visualizer.Variants.FREQUENCY,␣
 ↪log=activities_log)
pn_visualizer.view(gviz)
```

```
replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
 ↪it/s]
```



The generated model is really very simple.

It can be observed that the following transitions have not been linked to any place:

'Cupboard', 'Maindoor', 'Fridge' and 'Cabinet'

This is because probably if connected, loops of length one and length two (not allowed by the algorithm) would be formed.

The Petri Net model obtained:

- is not *Safeness* since *'microwave'* can infinitely increase tokens.
- is *deadlock-free* since it can always be fired *'microwave'*.

[35]:
```
#check_soundness
woflan.apply(net, initial_marking, final_marking)
```

```
Input is ok.
There is more than one source place.
```

[35]: False

By constructing the reachability graph I was able to find some possible sequences of actions that the model allows:

Bed –> Basin –> Cooktop –> Toilet –> Shower –> Toaster –> Seat –> ...

Bed –> Basin –> Toilet –> Cooktop –> Shower –> Toaster –> Seat –> ...

Bed –> Toilet –> Shower –> Basin –> Cooktop –> Toaster –> Seat –> ...

Bed –> Toilet –> Shower –> Basin –> Toaster –> Cooktop –> ...

Bed –> Toilet –> Shower –> Basin –> Toaster –> Seat –> Cooktop –> ...

Bed –> Toilet –> Shower –> Toaster –> Basin –> Cooktop

Bed –> Toilet –> Shower –> Toaster –> Basin –> Seat –> Cooktop –> ...

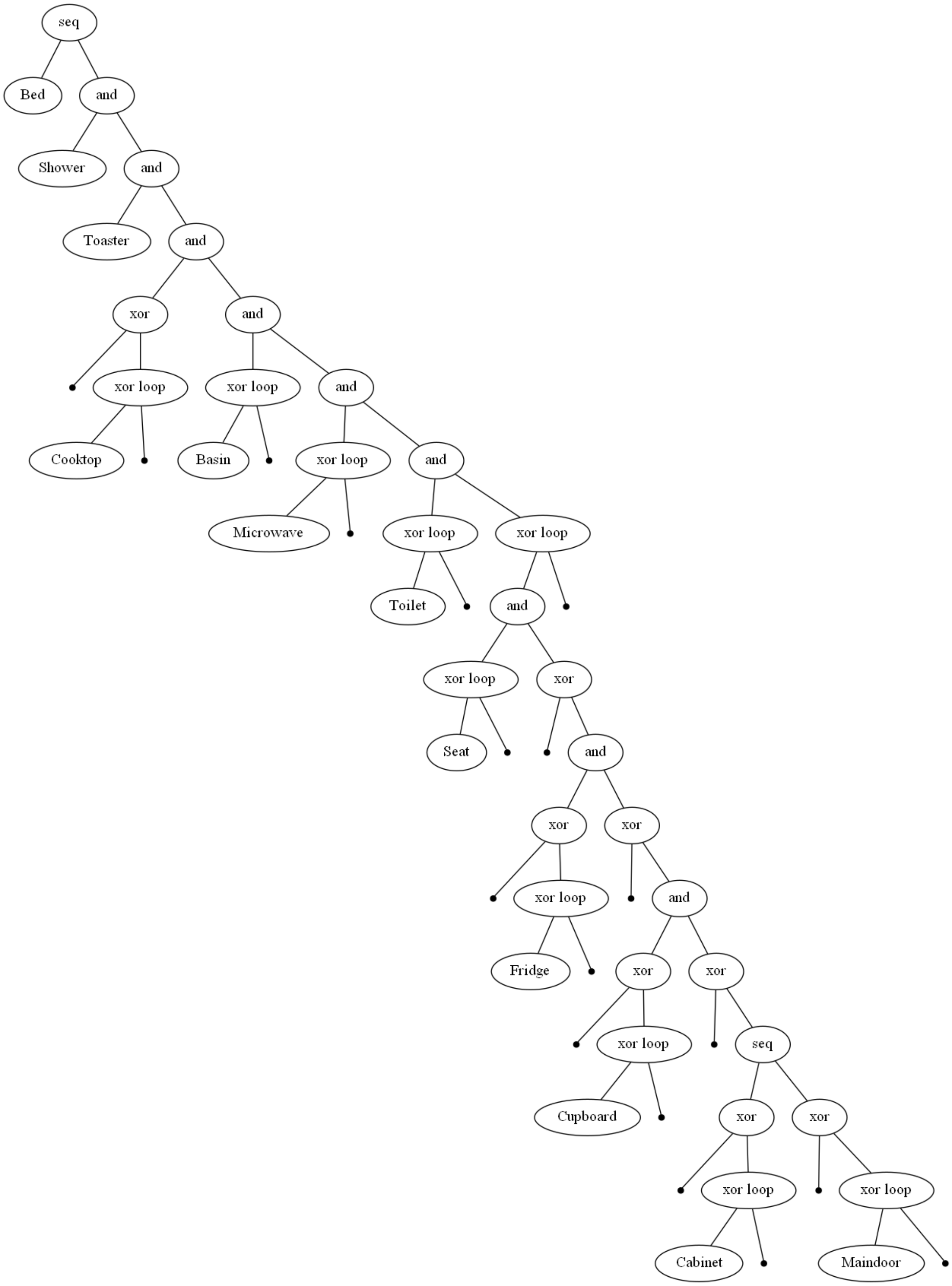
Bed –> Toilet –> Shower –> Toaster –> Seat –> Basin –> Cooktop –> ...

Bed –> Toilet –> Basin –> Cooktop –> Shower –> Toaster –> Seat –> ...

Heuristic Miner, Inductive Miner, DFG discovery & Tree Inductive

[ ]:
```
#create_and_save function
create_and_save_all_models(sensors_log,"Models/2.1/")
```

Let's take a look at one of the derived models as an example:



This process tree shows that the first sensor is the bed sensor, followed by any one of all the others. Where, while shower and toaster are always present, all the other sensors may not be present or may be repeated several times. The model reflects that of human activities.

6.5.2 (2.2) The sensor activiation model with respect to the individual rooms in which they are located.

In order to derive the model, I filtered the data by room obtaining new logs each containing the specific events for that room:

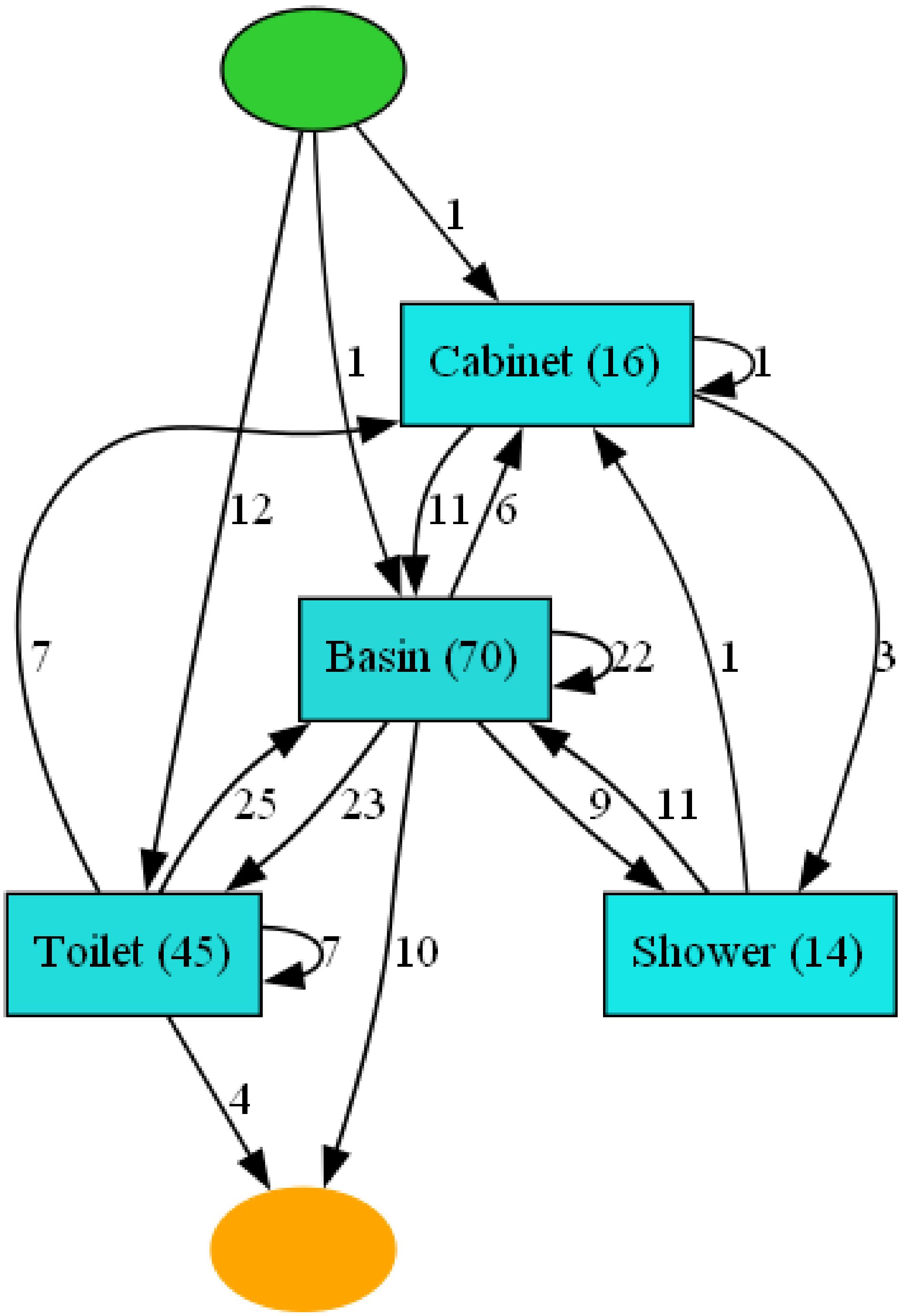
[37]:
```
slice_Bedroom      = pm4py.
 ↪filter_event_attribute_values(sensors_log,'Place',{'Bedroom'},    ␣
 ↪level='event', retain=True)
slice_Bathroom     = pm4py.
 ↪filter_event_attribute_values(sensors_log,'Place',{'Bathroom'},   ␣
 ↪level='event', retain=True)
slice_Kitchen      = pm4py.
 ↪filter_event_attribute_values(sensors_log,'Place',{'Kitchen'},    ␣
 ↪level='event', retain=True)
slice_Living       = pm4py.
 ↪filter_event_attribute_values(sensors_log,'Place',{'Living'},     ␣
 ↪level='event', retain=True)

Bedroom  = pm4py.convert_to_event_log(slice_Bedroom)
Bathroom = pm4py.convert_to_event_log(slice_Bathroom)
Kitchen  = pm4py.convert_to_event_log(slice_Kitchen)
Living   = pm4py.convert_to_event_log(slice_Living)
```

After that, by using a for loop, I derived the models for each log and saved them (in the folder: *'Models/2.2/'*).

[38]:
```
rooms_log = [Bedroom, Bathroom, Kitchen, Living]

for j in rooms_log:
    #
    #select Path
    folderName= namestr(j, globals())
    path="Models/2.2/"+folderName[0] + "/"
    print("saving models in: ", path)
    #
    #create_and_save function
    create_and_save_all_models(j,path)
```

```
saving models in:  Models/2.2/Bedroom/
Dfg dictionary not saved
saving models in:  Models/2.2/Bathroom/
saving models in:  Models/2.2/Kitchen/
saving models in:  Models/2.2/Living/
```

Let's take a look at one of the derived models as an example:



6.5.3 (2.3) The model describing the path executed by person A.

In order to derive the model, I personally selected which columns to use in the algorithms:

[40]:
```
#  Alpha inductive algorithm
petri_alpha, im_alpha, fm_alpha = pm4py.discover_petri_net_alpha (sensors_log,␣
↪activity_key='Place', timestamp_key='Start', case_id_key='case:concept:name')
```

```
#  Heuristic miner algorithm
petri_heuristics, im_heuristics, fm_heuristics = pm4py.
 ↪discover_petri_net_heuristics (sensors_log, dependency_threshold=0.0,␣
 ↪and_threshold=0.0, loop_two_threshold=1,  activity_key='Place',␣
 ↪timestamp_key='Start', case_id_key='case:concept:name')

#  Inductive miner algorithm
net2, im2, fm2 = pm4py.discover_petri_net_inductive (sensors_log, True, 0.0,␣
 ↪activity_key='Place', timestamp_key='Start', case_id_key='case:concept:name')

#  Tree inductive  algorithm
tree_inductive = pm4py.discover_process_tree_inductive (sensors_log, False, 0.
 ↪0, activity_key='Place', timestamp_key='Start', case_id_key='case:concept:
 ↪name')

#  Heuristic miner algorithm
heu_net = pm4py.discover_heuristics_net (sensors_log, dependency_threshold=0.5,␣
 ↪and_threshold=0.65, loop_two_threshold=0.5,  min_act_count=1,␣
 ↪min_dfg_occurrences=1, activity_key='Place', timestamp_key='Start',␣
 ↪case_id_key='case:concept:name')

#  dfg_discovery   algorithm
dfg, dfg_sa, dfg_ea = pm4py.discover_dfg (sensors_log, activity_key='Place',␣
 ↪timestamp_key='Start', case_id_key='case:concept:name')

path="Models/2.3/"
#Save models
pm4py.save_vis_petri_net(petri_alpha, im_alpha, fm_alpha, path+"alpha.png")
pm4py.save_vis_petri_net(petri_heuristics, im_heuristics, fm_heuristics,␣
 ↪path+"heuristics.png")
pm4py.save_vis_petri_net(net2, im2, fm2, path+"inductive.png")
pm4py.save_vis_process_tree(tree_inductive, path+"inductive_tree.png")
pm4py.save_vis_heuristics_net(heu_net, path+"heunet.png")
pm4py.save_vis_dfg(dfg, dfg_sa, dfg_ea, path+"dfg.png")
#Save models as Dictionary
pm4py.write_pnml(petri_alpha, im_alpha, fm_alpha, path+"Dictionary/alpha.pnml")
pm4py.write_pnml(petri_heuristics, im_heuristics, fm_heuristics,␣
 ↪path+"Dictionary/heuristics.pnml")
pm4py.write_pnml(petri_inductive, im_inductive, fm_inductive, path+"Dictionary/
 ↪inductive.pnml")
pm4py.write_ptml(tree_inductive, path+"Dictionary/inductive.ptml")
pm4py.write_dfg(dfg, dfg_sa, dfg_ea, path+"Dictionary/dfg.dfg")
```

[41]:
```
#check_soundness
woflan.apply(petri_alpha, im_alpha, fm_alpha)
```

```
Input is ok.
There is more than one source place.
```

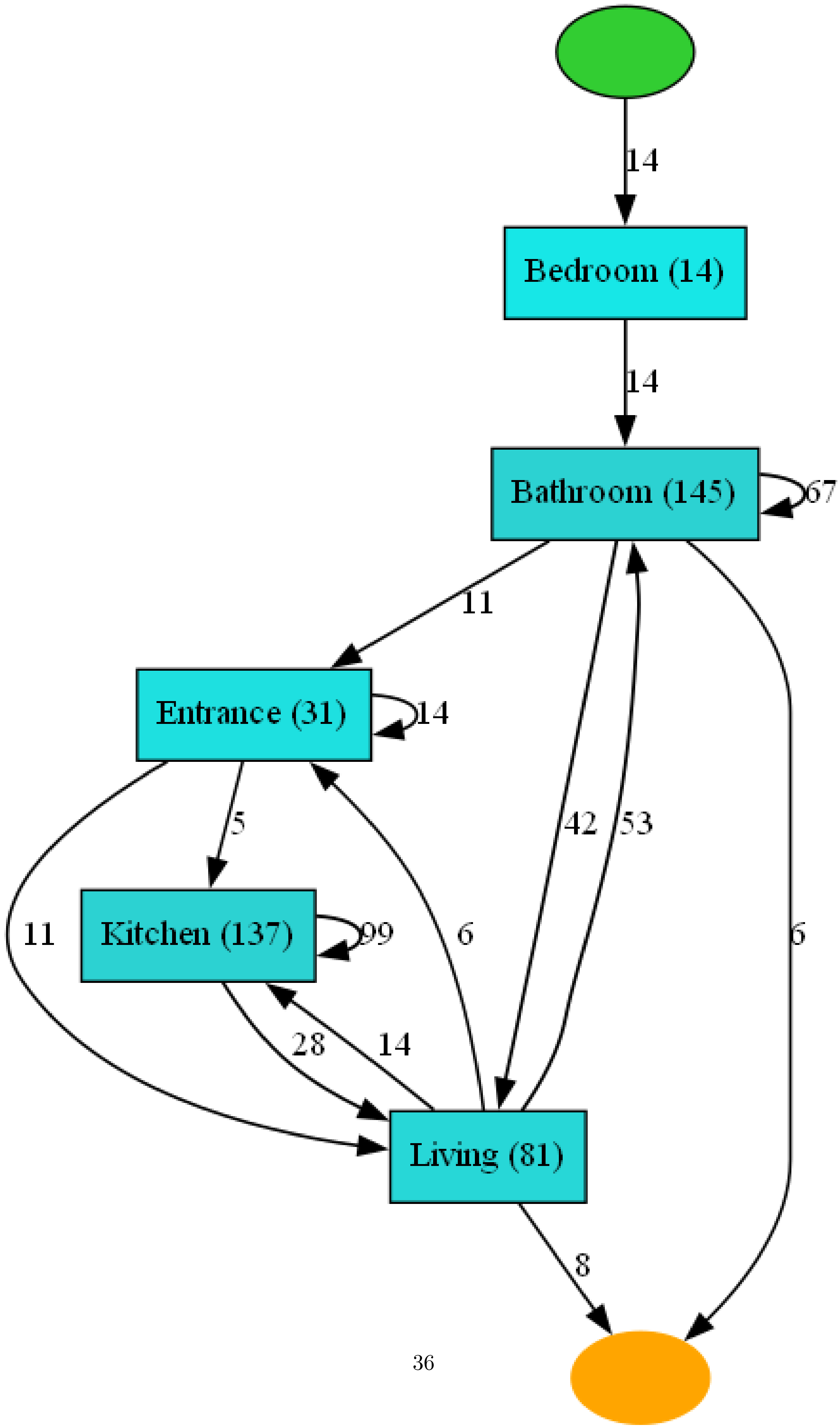
[41]: False

[42]:
```
#check_soundness
woflan.apply(net2, im2, fm2)
```

```
Input is ok.
Petri Net is a workflow net.
Every place is covered by s-components.
There are no dead tasks.
All tasks are live.
```

[42]: True

Let's take a look at one of the derived models as an example:



14
Bedroom (14)
14
Bathroom (145)
67
11
Entrance (31)
14
5
42
53
Kitchen (137)
99
11
6
6
28
14
Living (81)
8

From this model it is possible to obtain the following information:

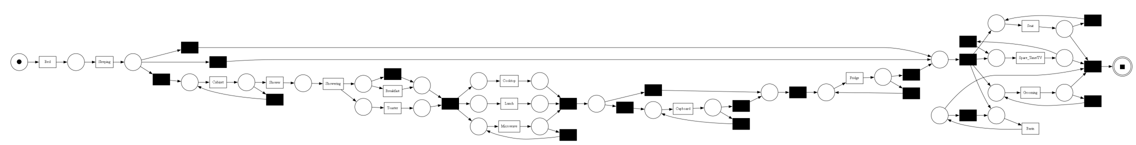
- After waking up the person A, he goes first to the bathroom.
- When he returns home after going out he goes directly to the the kitchen or in the living room.
- Very often it goes from the slot to the bathroom and then return to the living room.
- From the kitchen after having lunch or breakfast always goes to the living room to spend free time.
- The last room where he performed actions before going to sleep was always the living room or the bathroom.

6.6 (3) Process Discovery on the all_events_log

6.6.1 (3.1) The model of daily human activities and sensors with which the person interacted.

[44]:
```
#  Alpha inductive algorithm
petri_alpha, im_alpha, fm_alpha = pm4py.discover_petri_net_alpha␣
↪(all_events_log, activity_key='Location', timestamp_key='Start',␣
↪case_id_key='case:concept:name')

#  Heuristic miner algorithm
petri_heuristics, im_heuristics, fm_heuristics = pm4py.
↪discover_petri_net_heuristics (all_events_log, dependency_threshold=0.6,␣
↪and_threshold=0.0, loop_two_threshold=1,  activity_key='Location',␣
↪timestamp_key='Start', case_id_key='case:concept:name')

#  Inductive miner algorithm
net2, im2, fm2 = pm4py.discover_petri_net_inductive(all_events_log, True, 0.6,␣
↪activity_key='Location', timestamp_key='Start', case_id_key='case:concept:
↪name')

#  Tree inductive  algorithm
tree_inductive = pm4py.discover_process_tree_inductive (all_events_log, True, 0.
↪6, activity_key='Location', timestamp_key='Start', case_id_key='case:concept:
↪name')

#  Heuristic miner algorithm
heu_net = pm4py.discover_heuristics_net (all_events_log, dependency_threshold=0.
↪6, and_threshold=0.65, loop_two_threshold=0.5,  min_act_count=1,␣
↪min_dfg_occurrences=1, activity_key='Location', timestamp_key='Start',␣
↪case_id_key='case:concept:name')

#  dfg_discovery   algorithm
```

```
dfg, dfg_sa, dfg_ea = pm4py.discover_dfg (all_events_log,␣
↪activity_key='Location', timestamp_key='Start', case_id_key='case:concept:
↪name')

path="Models/3.1/"
#Save models
pm4py.save_vis_petri_net(petri_alpha, im_alpha, fm_alpha, path+"alpha.png")
pm4py.save_vis_petri_net(petri_heuristics, im_heuristics, fm_heuristics,␣
↪path+"heuristics.png")
pm4py.save_vis_petri_net(net2, im2, fm2, path+"inductive.png")
pm4py.save_vis_process_tree(tree_inductive, path+"inductive_tree.png")
pm4py.save_vis_heuristics_net(heu_net, path+"heunet.png")
pm4py.save_vis_dfg(dfg, dfg_sa, dfg_ea, path+"dfg.png")
#Save models as Dictionary
pm4py.write_pnml(petri_alpha, im_alpha, fm_alpha, path+"Dictionary/alpha.pnml")
pm4py.write_pnml(petri_heuristics, im_heuristics, fm_heuristics,␣
↪path+"Dictionary/heuristics.pnml")
pm4py.write_pnml(petri_inductive, im_inductive, fm_inductive, path+"Dictionary/
↪inductive.pnml")
pm4py.write_ptml(tree_inductive, path+"Dictionary/inductive.ptml")
pm4py.write_dfg(dfg, dfg_sa, dfg_ea, path+"Dictionary/dfg.dfg")

pm4py.view_petri_net(net2, im2, fm2)
```



[45]:
```
#check_soundness
woflan.apply(petri_alpha, im_alpha, fm_alpha)
```

```
Input is ok.
There is more than one source place.
```

[45]: False

[46]:
```
#check_soundness
woflan.apply(net2, im2, fm2)
```

```
Input is ok.
Petri Net is a workflow net.
Every place is covered by s-components.
There are no dead tasks.
All tasks are live.
```

[46]: True

6.7 Model checking

With *PM4Py* it is also possible to apply algorithms to check the correspondence of a model to some logs.

These algorithms are able not only to show the result but also the diagnostics, i.e. the process that led to that result. In particular, it allows one to see whether each trace has been fully covered by the model or whether there are skipped or missing tokens.

So I decided first of all, to verify the accuracy of each model for the log that generated it. To then use different logs from the original to understand if they are compliant with the models.

6.7.1 Compliance of each model for the original log (log of A)

Compliance check on the Alpha model

[47]:
```
# Conformance cheching
petri_alpha, im_alpha, fm_alpha = pm4py.read_pnml("Models/1.1/Dictionary/alpha.
↪pnml")
tbr_diagnostics = pm4py.
↪conformance_diagnostics_token_based_replay(activities_log, petri_alpha,␣
↪im_alpha, fm_alpha)
info            = pm4py.fitness_token_based_replay(activities_log, petri_alpha,␣
↪im_alpha, fm_alpha)
df_info = pd.DataFrame([info])
#Show info Token_based_replay
df_info.T
```

```
replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
↪it/s]

replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
↪it/s]
```

[47]:

	0
perc_fit_traces	0.000000
average_trace_fitness	0.592086
log_fitness	0.585106
percentage_of_fitting_traces	0.000000

[48]:
```
pd.set_option('display.max_rows', 14)                  #max number of lines that␣
↪can be displayed
# show diagnostics (for each trace / case)
df_diagnostics = pd.DataFrame(tbr_diagnostics)
df_diagnostics['activated_transitions'] = [len(_) for _ in df_diagnostics.
↪activated_transitions]
df_diagnostics.iloc[:, [0, 1, 2, 6, 7, 8, 9]]         #useful columns
```

[48]:

```
    trace_is_fit  trace_fitness  activated_transitions  missing_tokens  \
0          False       0.666667                     15               4
1          False       0.562500                     23               7
2          False       0.538462                     15               6
3          False       0.571429                     19               6
4          False       0.583333                     13               5
5          False       0.700000                     10               3
6          False       0.642857                     19               5
7          False       0.466667                     23               8
8          False       0.571429                     19               6
9          False       0.636364                     16               4
10         False       0.636364                     13               4
11         False       0.615385                     19               5
12         False       0.526316                     25               9
13         False       0.571429                     19               6

    consumed_tokens  remaining_tokens  produced_tokens
0                12                 4               12
1                16                 7               16
2                13                 6               13
3                14                 6               14
4                12                 5               12
5                10                 3               10
6                14                 5               14
7                15                 8               15
8                14                 6               14
9                11                 4               11
10               11                 4               11
11               13                 5               13
12               19                 9               19
13               14                 6               14
```

Compliance check on the Inductive model

[49]:

```python
# Conformance cheching
net, im, fm = pm4py.read_pnml("Models/1.1/Dictionary/inductive.pnml")
tbr_diagnostics = pm4py.
 ↪conformance_diagnostics_token_based_replay(activities_log, net, im, fm)
info             = pm4py.fitness_token_based_replay(activities_log, net, im, fm)
df_info = pd.DataFrame([info])
#Show info Token_based_replay
df_info.T
```

```
replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
 ↪it/s]

replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
 ↪it/s]
```

[49]:
```
                                  0
perc_fit_traces               100.0
average_trace_fitness           1.0
log_fitness                     1.0
percentage_of_fitting_traces  100.0
```

[50]:
```
# show diagnostics (for each trace / case)
df_diagnostics = pd.DataFrame(tbr_diagnostics)
df_diagnostics['activated_transitions'] = [len(_) for _ in df_diagnostics.
 ↪activated_transitions]
df_diagnostics.iloc[:, [0, 1, 2, 6, 7, 8, 9]]        #useful columns
```

[50]:
```
    trace_is_fit  trace_fitness  activated_transitions  missing_tokens  \
0           True            1.0                     26               0
1           True            1.0                     42               0
2           True            1.0                     28               0
3           True            1.0                     36               0
4           True            1.0                     24               0
5           True            1.0                     16               0
6           True            1.0                     34               0
7           True            1.0                     42               0
8           True            1.0                     34               0
9           True            1.0                     28               0
10          True            1.0                     24               0
11          True            1.0                     36               0
12          True            1.0                     46               0
13          True            1.0                     34               0

    consumed_tokens  remaining_tokens  produced_tokens
0                34                 0               34
1                50                 0               50
2                36                 0               36
3                44                 0               44
4                32                 0               32
5                24                 0               24
6                42                 0               42
7                50                 0               50
8                42                 0               42
9                36                 0               36
10               32                 0               32
11               44                 0               44
12               54                 0               54
13               42                 0               42
```

Compliance check on the Heuristics model

[51]:
```
# Conformance cheching
net, im, fm = pm4py.read_pnml("Models/1.1/Dictionary/heuristics.pnml")
```

```
tbr_diagnostics = pm4py.
  ↪conformance_diagnostics_token_based_replay(activities_log, net, im, fm)
info            = pm4py.fitness_token_based_replay(activities_log, net, im, fm)
df_info = pd.DataFrame([info])
#Show info Token_based_replay
df_info.T
```

```
replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
 ↪it/s]

replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
 ↪it/s]
```

[51]:

	0
perc_fit_traces	0.000000
average_trace_fitness	0.925501
log_fitness	0.926968
percentage_of_fitting_traces	0.000000

[52]:

```
# show diagnostics (for each trace / case)
df_diagnostics = pd.DataFrame(tbr_diagnostics)
df_diagnostics['activated_transitions'] = [len(_) for _ in df_diagnostics.
  ↪activated_transitions]
df_diagnostics.iloc[:, [0, 1, 2, 6, 7, 8, 9]]        #useful columns
```

[52]:

	trace_is_fit	trace_fitness	activated_transitions	missing_tokens \
0	False	0.939167	21	1
1	False	0.909244	35	3
2	False	0.898333	21	2
3	False	0.892519	28	3
4	False	0.878571	16	2
5	False	0.904167	13	1
6	False	0.923295	29	2
7	False	0.913720	37	3
8	False	0.925579	30	2
9	False	0.982759	26	0
10	False	0.930736	18	1
11	False	0.955437	30	1
12	False	0.917774	38	3
13	False	0.985714	32	0

	consumed_tokens	remaining_tokens	produced_tokens
0	24	2	25
1	38	4	39
2	24	3	25
3	32	4	33
4	20	3	21
5	15	2	16

6	32	3	33
7	40	4	41
8	33	3	34
9	28	1	29
10	21	2	22
11	33	2	34
12	42	4	43
13	34	1	35

As can be seen, the *'alpha.pnml'* model is not compliant with the original log and therefore does not represent the process of the activities performed by person A during his day.

The same for *'heuristics.pnml'* but it is much closer to achieving compliance. In contrast, *'inductive.pnml'* is compliant and therefore fully represents the pattern of Person A's daily activities. Perhaps, however, this model is too permissive.

6.7.2 Compliance of the model for the log of B

Let's try now to test the model with the log of person B, to see if the algorithm recognises the activities as compliant or is able to recognise that they do not belong to person A.

Compliance check on the Inductive model

[55]:
```
# Conformance cheching
net, im, fm = pm4py.read_pnml("Models/1.1/Dictionary/inductive.pnml")
tbr_diagnostics = pm4py.
 ↪conformance_diagnostics_token_based_replay(activities_log_B, net, im, fm)
info            = pm4py.fitness_token_based_replay(activities_log_B, net, im,␣
 ↪fm)
df_info = pd.DataFrame([info])
#Show info Token_based_replay
df_info.T
```

```
replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
 ↪it/s]

replaying log with TBR, completed variants ::   0%|          | 0/14 [00:00<?, ?
 ↪it/s]
```

[55]:

	0
perc_fit_traces	100.0
average_trace_fitness	1.0
log_fitness	1.0
percentage_of_fitting_traces	100.0

[56]:
```
# show diagnostics (for each trace / case)
df_diagnostics = pd.DataFrame(tbr_diagnostics)
df_diagnostics['activated_transitions'] = [len(_) for _ in df_diagnostics.
 ↪activated_transitions]
df_diagnostics.iloc[:, [0, 1, 2, 6, 7, 8, 9]]        #useful columns
```

[56]:
```
    trace_is_fit  trace_fitness  activated_transitions  missing_tokens  \
0           True            1.0                     26               0
1           True            1.0                     42               0
2           True            1.0                     28               0
3           True            1.0                     36               0
4           True            1.0                     24               0
5           True            1.0                     16               0
6           True            1.0                     34               0
7           True            1.0                     42               0
8           True            1.0                     34               0
9           True            1.0                     28               0
10          True            1.0                     24               0
11          True            1.0                     36               0
12          True            1.0                     46               0
13          True            1.0                     34               0

    consumed_tokens  remaining_tokens  produced_tokens
0                34                 0               34
1                50                 0               50
2                36                 0               36
3                44                 0               44
4                32                 0               32
5                24                 0               24
6                42                 0               42
7                50                 0               50
8                42                 0               42
9                36                 0               36
10               32                 0               32
11               44                 0               44
12               54                 0               54
13               42                 0               42
```

In this test, *'inductive.pnml'*, it is compliant again. Therefore, the model is not able to recognise, as extraneous, activities performed by another person. So the model is probably suitable for recognising human activities in general or simply allows any action even innatuaral.

To ensure this, we can test the model with a log containing improbable traces.

6.7.3 Compliance of each model for a very unrealistic log

Now let us see if for the model created, is recognised a log with unrealistic activities, times and durations. To do this, it was first necessary to create a dataset as a list. Then I converted the list into a dataframe and formatted it in order to apply the log transformation.

[57]:
```
#Create a list of events to test
unrealistic_activity_list = [['2011-11-28 15:00:00+00:00', '2011-11-28 20:00:
↪00+00:00','Sleeping','0'],
```

```
       ['2011-11-28 20:01:00+00:00', '2011-11-28 21:00:00+00:
↪00','Leaving','0'],
       ['2011-11-28 21:02:00+00:00', '2011-11-28 21:07:00+00:00','Snack','0'],
       ['2011-11-28 21:09:00+00:00', '2011-11-28 22:00:00+00:
↪00','Breakfast','0'],
       ['2011-11-28 22:12:00+00:00', '2011-11-28 22:18:00+00:00','Lunch','0'],
       ['2011-11-28 22:20:00+00:00', '2011-11-28 22:30:00+00:
↪00','Toileting','0'],
       ['2011-11-28 22:31:00+00:00', '2011-11-28 22:40:00+00:
↪00','Showering','0'],
       ['2011-11-28 22:41:00+00:00', '2011-11-28 23:55:00+00:00','Spare_Time/
↪TV','0'],
       ['2011-11-29 00:01:00+00:00', '2011-11-29 00:40:00+00:
↪00','Toileting','0'],
       ['2011-11-29 01:00:00+00:00', '2011-11-29 12:40:00+00:
↪00','Leaving','0'],
       ['2011-11-29 13:00:00+00:00', '2011-11-29 14:45:00+00:00','Spare_Time/
↪TV','0'],
       ['2011-11-28 15:00:00+00:00', '2011-11-28 20:00:00+00:
↪00','Sleeping','1'],
       ['2011-11-28 20:01:00+00:00', '2011-11-28 21:00:00+00:00','Spare_Time/
↪TV','1'],
       ['2011-11-28 21:02:00+00:00', '2011-11-28 21:07:00+00:00','Snack','1'],
       ['2011-11-28 21:09:00+00:00', '2011-11-28 22:00:00+00:
↪00','Leaving','1'],
       ['2011-11-28 22:12:00+00:00', '2011-11-28 22:18:00+00:
↪00','Showering','1'],
       ['2011-11-28 22:20:00+00:00', '2011-11-28 22:30:00+00:
↪00','Toileting','1'],
       ['2011-11-28 22:31:00+00:00', '2011-11-28 22:40:00+00:00','Lunch','1'],
       ['2011-11-28 22:41:00+00:00', '2011-11-28 23:55:00+00:00','Spare_Time/
↪TV','1'],
       ['2011-11-29 00:01:00+00:00', '2011-11-29 00:40:00+00:
↪00','Toileting','1'],
       ['2011-11-29 01:00:00+00:00', '2011-11-29 12:40:00+00:
↪00','Leaving','1'],
       ['2011-11-29 13:00:00+00:00', '2011-11-29 14:45:00+00:00','Snack','1']]
```

[68]:

```
#Create the pandas DataFrame from the list
unrealistic_df = pd.DataFrame(unrealistic_activity_list, columns=['Start',␣
↪'End','Activity','LifeCycles'])
unrealistic_df['LifeCycles'] = unrealistic_df['LifeCycles'].astype("object")
unrealistic_df['Start']      = unrealistic_df['Start'].astype("datetime64")
unrealistic_df['End']        = unrealistic_df['End'].astype("datetime64")
#DataFrame formatted for pm4py
```

```
unrealistic_df=pm4py.format_dataframe(unrealistic_df, case_id="LifeCycles",␣
↪activity_key="Activity", timestamp_key="Start")
#DataFrame convert into log
unrealistic_log= pm4py.convert_to_event_log(unrealistic_df)
```

Compliance check on the Inductive model

[61]:
```
# Conformance cheching
net, im, fm = pm4py.read_pnml("Models/1.1/Dictionary/inductive.pnml")
tbr_diagnostics = pm4py.
↪conformance_diagnostics_token_based_replay(unrealistic_log, net, im, fm)
info            = pm4py.fitness_token_based_replay(unrealistic_log, net, im, fm)
df_info = pd.DataFrame([info])
#Show info Token_based_replay
df_info.T
```

```
replaying log with TBR, completed variants ::   0%|          | 0/2 [00:00<?, ?it/
↪s]

replaying log with TBR, completed variants ::   0%|          | 0/2 [00:00<?, ?it/
↪s]
```

[61]:

	0
perc_fit_traces	0.000000
average_trace_fitness	0.823345
log_fitness	0.823416
percentage_of_fitting_traces	0.000000

[62]:
```
# Result (for each trace / case)
df_diagnostics = pd.DataFrame(tbr_diagnostics)
df_diagnostics['activated_transitions'] = [len(_) for _ in df_diagnostics.
↪activated_transitions]
df_diagnostics.iloc[:, [0, 1, 2, 6, 7, 8, 9]]#useful columns
```

[62]:

	trace_is_fit	trace_fitness	activated_transitions	missing_tokens \
0	False	0.819838	18	1
1	False	0.826852	19	1

	consumed_tokens	remaining_tokens	produced_tokens
0	19	8	26
1	20	8	27

As can be seen from the tables showing the results, the log with dummy data is not considered to be compliant with the model. Therefore, the model can be used to recognise realistic human daily activities.

7 Conclusion

The stuido case required three steps.

The first phase consisted of **studying the logs** to understand the data and their useful information needed to understand which models to create and how to do so.

I found:

- that the different datasets are related to each other.
- the number of traces and the different actions.
- the different variants of actions. (unique sequences)
- the distribution of events over various periods taken into account (hours/days per month/days per week)

The second phase was **Process Discovery**. Algorithms provided by *PM4Py* were used to derive useful models to describe the relationship of the data.

Types of models created are:

- 1.1) The model of daily activities (of person A) over the entire time interval considered.
- 1.2) The model of daily activities (of person A) on individual days of the week.
- 2.1) The sensor activiation model with respect to the entire time.
- 2.2) The sensor activiation model with respect to the individual rooms in which they are located.
- 2.3) The model describing the path executed by person A.
- 3.1) The model of daily human activities and sensors with which the person interacted.

These models were saved in the *'Models'* folder. In particular, each model type indicated by a number was saved in the folder named with the same number. (e.g. *"Models/1.1/model.png"*) For each model type, several models were generated by applying different algorithms.

Of these models, those represented as a Petri Net were analysed by determining the following properties:

- *Safeness*
- *Deadlock-free*
- *Soundness*

The third and final step was **Model Checking**. First of all, I checked the suitability of the models to the log that generated them. I discovered that the best model, from this point of view, is the one generated by the inductive algorithm. Subsequently, I checked whether this model was representative only of the activities performed by person A. But I discovered that it is not.

Finally, I tested the model with unsuitable tracks as they are unrealistic, and discovered that they are not compliant. This implies that while the model is not able to distinguish the activities of one person from those of another, it is able to recognise the realisability of the logs.