# CSM: Requirement Analysis

Odysseas Karanikas
odysseas.karanikas@rwth-aachen.de

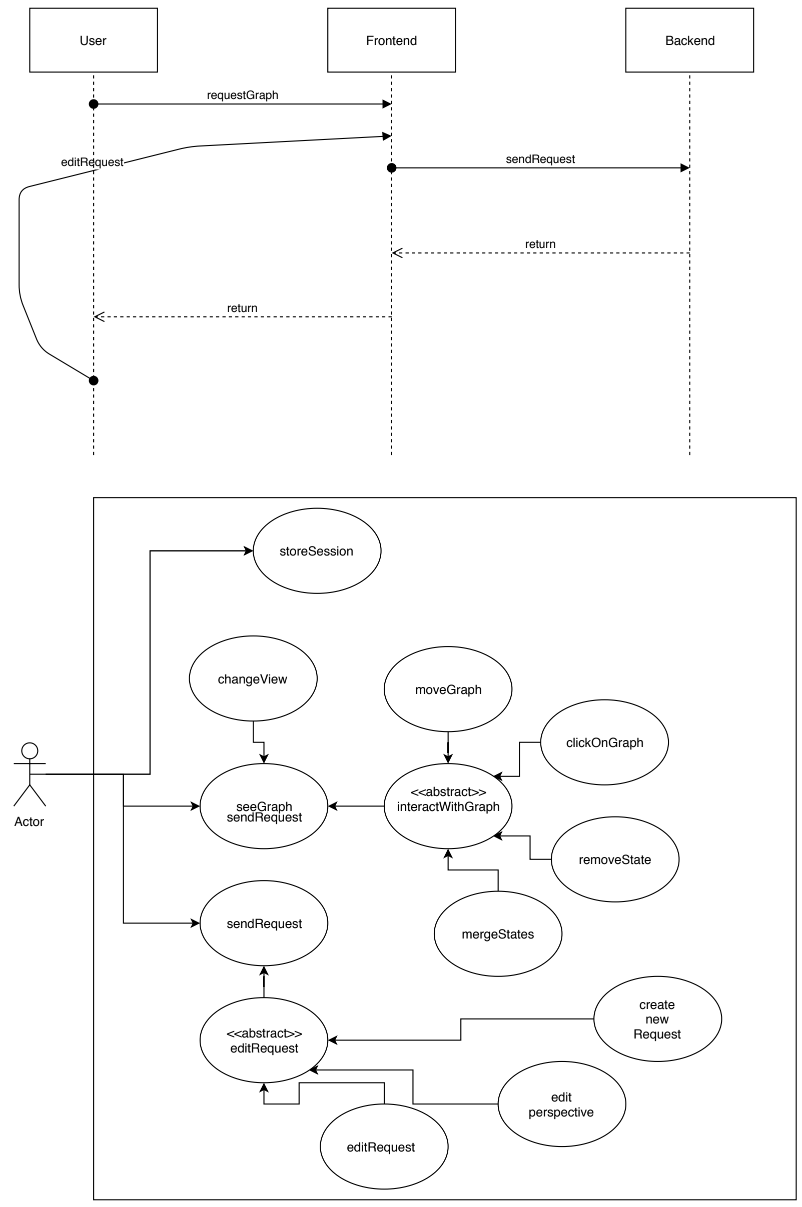
Mann, Daniel
daniel.mann@rwth-aachen.de

Daniel Rein
drein99@outlook.de

April 2019

## 1 Introduction

In the following document, the general functions of the CSM Miner web application will be outlined.
After connecting to the web application, the user can copy his event log into a textbox where he is able to define labels for the different states of the resulting graphs. When every label is defined, the user can upload the modified log to the backend, where the graphs are computed and returned to the web application, providing crucial information for transitions and states which is displayed after the user clicks on the graph.
On the web application, the graphs will be displayed and the user can then interact with them. He or she will be able to drag the graphs to reposition them, zoom in and out, click on states and transitions to get information and switch between the graphs of the different perspectives.
After clicking on a transition or state, additionally to the information mentioned above, the related states and transitions are highlighted in the other views aswell. Every time the user edits the graph, the edited request will be transmitted to the backend, where the changes are interpreted and the new resulting graph will be displayed on the web application.
After the basic project is finished, combining multiple states to one merged state and delete states that are not needed in practice can be implemented. This is all visualized in the diagrams in figure 1.



**(a)**

**Figure 1:** *A sequence diagram and an use case diagram both describing the interaction between the application and the user.*

## 2 Input/Output Specification

The planned software can be understood as a mapping of a given input to an output. In our case, the input is an XES log file and one or more views and the output is a state chart. Since this function is our core feature it is important to discuss the details of input and output.

### 2.1 Input

The initial input of our software will be an XES file. XES stands for extensible event stream. It is a standard format for the representation of causally linked and ordered events and makes use of the XML format [**xes**]. Generally, the structure of an XES file includes a trace and an event element where an event is a subelement of a trace. An event can contain attributes such as a timestamp, a name and other individually specified properties. An excerpt of an XES file can look as follows[1]:

```
<trace>
    <string key="concept:name"
        value="Trace number one"/>
    <event>
        <string key="concept:name"
            value="Register client"/>
        <string key="system" value="alpha" />
        <date key="time:timestamp"
            value="2009-11-25T14:12:45:000+02:00" />
        <int key="attempt" value="23">
            <boolean key="tried hard" value="false" />
        </int>
    </event>
    <event>
        <string key="concept:name"
            value="Mail rejection"/>
        <string key="system" value="beta" />
        <date key="time:timestamp"
            value="2009-11-28T11:18:45:000+02:00" />
    </event>
</trace>
```

[1]Usually an XES file contains additional metadata and definition of elements and attributes which we omitted in our example.

A trace represents a closed and ordered process in the system while events depict stages or key points in the process. For example in an insurance company, one could consider all milestones of an insurance case as events belonging to the same trace. Another case would then necessitate another trace.

In our case the event log will have a particular semantic. We will understand the events as states that a process goes through where these states are spread in time. That is every event has a duration attribute denoting how long a given process, represented by a trace, spent in the corresponding event. To make this clearer, we will take the example of a grocery store. A trace and thus a process corresponds to a customer pathing through the store. On his way through the store he will go through different locations which will be his states. These locations can be the entrance ($S$), two different sections of the bakery ($B_1$ and $B_2$), two different subsections of the beverage section ($G_1$, $G_2$) and finally the cash register ($P$) and the exit section $E$. A possible log file can look as follows:

```
Trace    State    Duration
1        S        1
1        B_1      3
1        G_2      4
1        P        5
1        E        1

2        S        1
2        G_1      3
2        P        4
2        E        1

3        S        1
3        E        1

4        S        1
4        B_2      3
4        P        3
4        E        1
```

Here we chose a different more simple representation that captures the order of states, their respective duration and their affiliated trace.

The log file is not the only input we expect. The user should furthermore be able to input a grouping as discussed in the Use case section. A grouping is an explicitly non-injective mapping from the initial state space to the target state space. We use the example of the grocery store from above and give two example groupings:

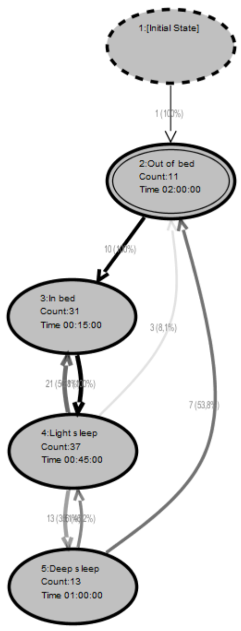
```
States   Grouping1       Grouping2

S        S        S'
B_1      B        S'
B_2      B        S'
G_1      G        G
G_2      G        G
P        E'       E'
E        E'       E'
```

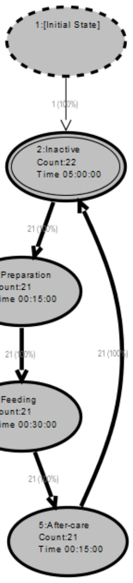
## 2.2 Output

We shall now specify in more detail what the output of our software should look like. For that we mostly require the same behaviour as the already existent and deployed software tool ProM [**prom**].

On a given input, that is, a log and an appropriate grouping, the user should obtain a view for each group of states that she determined or that came predetermined with the log file. Each view contains a graph of states and their transitions. Here, we require the states to contain statistics about the number of its occurrences and optionally their average duration times. We require the transitions to be annotated with the relative frequency of their occurrence. That is, if the source state occurs $N$ times and it is directly followed in $M$ of these $N$ times, then we annotate this transition from source to target state with the number $M/N$. Each view is supposed to contain only states from one of the groups chosen prior. An example of such a graphical output is shown in figure 2.



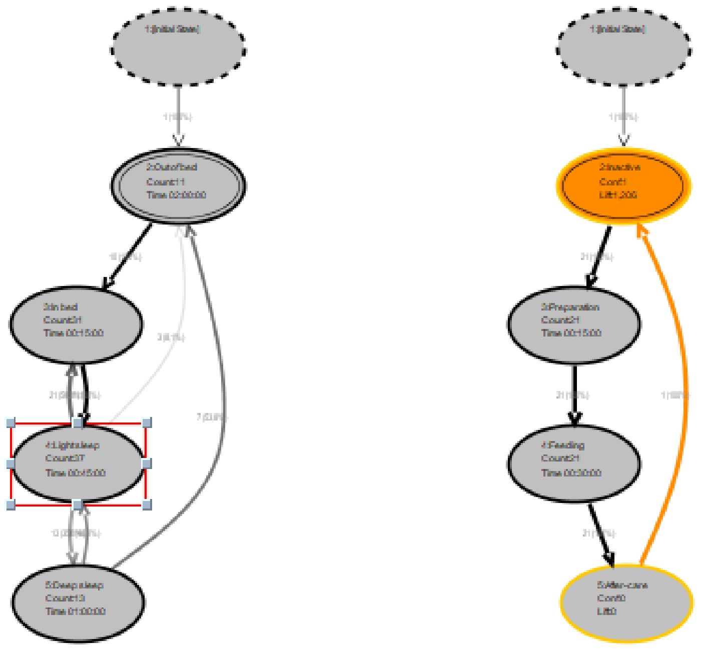
**(a)**



**(b)**

**Figure 2:** *An example output for a system that is partitioned in a sleeping (2a) and a feeding (2a) cycle. The states and transitions are annotated with the statistics described in the text. The output is computed with the CSM Miner provided by the ProM tool.*

Generally, the output graphs should also illustrate the interactions between the views. A requirement taken from this might be that a click of a state in a view should mark the co-occurring states in the other views and annotate them with additional statistics of their relative frequency of co-occurrence conditioned on the number of occurrences of the clicked state. This functionality is implemented by the ProM CSM Miner. An example of such a click functionality we find in figure 3.



**(a)** **(b)**

**Figure 3:** *An example click behaviour for the system introduced as an example above. In the feeding view (3a) the state "light sleep" has been clicked. This leads to the highlighting of the two co-occurring states "inactive" and "after-care" in the feeding view (3b). Also shown is information about the relative frequency of co-occurrence conditioned on the frequency of the clicked state.*

A consultation of the advisor/client has resulted in another requirement on the click behaviour: A click in one view will result in the marking of direct and indirect successors in the other views and every such marked state is again annotated with appropriate relative frequencies.

## 3 Technical Requirement

### 3.1 Needed software

To deploy the solution there are multiple requirements. Firstly a python version greater 3 is needed. Next there is pip the software that will manage our server application. Now all we need server-wise is Django , the project was written under Django 2.2 , therefore it is recommended to use at least version 2.2.

## 3.2 Client-Server architecture

The solution requires a rather extensive client-server architecture for computational and quality of service reasons. Since the main computation is on the server the client would normally need to refresh every time a request is sent. This is not optimal for a mobile work environment since wireless networks are not as consistent as wired ones. Therefor the software only causes the client to do a complete refresh once a major computation needs to be done (e.g. create a new view), but smaller requests (e.g. labeling in current view) are computed client side. Since a project is not stored locally we will implement a request queue that will be used to handle the asynchronous work load. This queue also ensures, that a request is never lost, because an element is not deleted before the corresponding confirmation is received.

## 3.3 Django management

To connect the front-end with the back-end Django will be used as a web server. The main tasks consist out of the following :

- Receive requests and files and sent them to the back-end.
- Sent processed information to the client in form of a http request.