Software Design Specification

Label Refinement by Behavioral Similarity

Document owners:
Bianka Bakullari
Christopher Beine
Nicole Ventsch
Juan Garza

Last edited: May 12, 2019

1 Introduction

1.1 System Overview

1.2 Design Map

1.3 Supporting Materials

1.4 Definitions and Acronyms

2 Design Considerations

2.1 Assumptions and Dependencies

In the design of our algorithm we will try to achieve the principles of a good design, such as loose coupling, high cohesion and data hiding. For each step of the algorithm we will use the least complex data structure depending on the input needed for the step. The trace variants of the input event log are part of the input of many steps of the algorithm. There is a lot of redundancy when it comes to the variants as input since sometimes the whole set must be considered, sometimes only pairs and other times both of these. We have decided to use dictionaries to relate the traces to their case IDs, and a network to save the mapping costs between each two traces. This part may still be prone to changes in case of other more efficient solutions.

2.2 Development Methods

There are still two problems which have to be discussed during implementation:

The first one has to do with the computation of the possible mappings. An exhaustive computation of all mappings and their corresponding costs is rather systematical and safe, whereas a Greedy approach where we order the event labels based on the frequency of their contexts may be more efficient.
Secondly, we use an undirected weighted graph to represent all events of all traces. It remains to be seen which method is more efficient in order to access each event based on the trace variant it belongs to. We could do this by refering to each event through their index in the network, or we could priorly use an injective mapping from all events to their labels.

3 Architectural Strategies

3.1 Sotware

As mentioned the Software is divided into two main Parts, the Front-end and Back-end.

The Back-end will be implemented with Python. Python is a popular and widely used programming language within the field of data science. It especially provides the pm4py library, that supports process mining algorithms. The webframework Flask which is written in Python too, provides a fast applicaton deployment with a minimal usage of external libraries so that we can focus on the algorithm design. A User can access the Back-end via an API which distributes the data to the required components.

The Front-end will be implemented with JavaScript. We will use a minimalistic Model-View-Controller (MVC) approach. The Model only represents the data like the customization paramters for the algorithm or the event log file. The Model will be easily converted into a JSON Object, which can be evaluated by the backend. The View represents the User Interface markup and is written in HTML5. Logic implementation is not required in the View part. The Controller part represents the business logic. Since the front-end acts as an abstraction layer for our Back-end API, the Controller should be as small as possible.

A database system is not set up, since the project focus is the refining algorithm. However, the application will keep track of the last five uploaded files and their refined event logs. For this we will use the server file system.

3.2 Future Enhancements

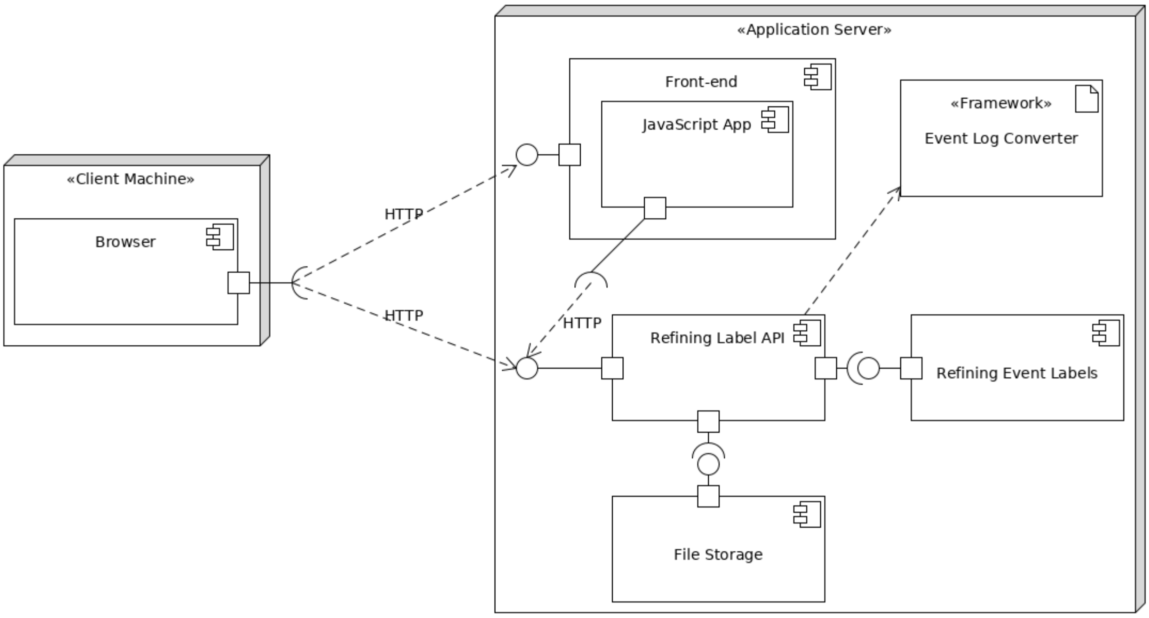
In order to realise a future development the following measure will be implemented: the *abstract class* will be used in combination with the *factory* pattern to exchange important software and algorithms components. This will be implemented for the file conversion, file creation or cost function. Additional functionality has to

implement a given interface which will be registerd to a factory method afterwards. Additional code is not required.
Developing an API which can be included into other system or workflow. The event log refinement can included into other applications. Individual components can be exchanged more easily.

4 System Architecture

The following UML displays the main application components and where they are deployed. The final application will be deployed on an application server which the user can access over an internet browser on his own machine. Therefore the system is independent of the users operating system. There are two ways to perform the algorithm:

- The user can access the JavaScript Front-end application via HTTP which provides a user friendly UI. There the user can set all required parameters and send them to the API.
- Additional, the user can access the API directly via HTTP with the required parameter. So the application can integrate into another application which could be helpful to perform preprocessing steps or to use the refined labels in another application.



4.1 Client Machine

The Client Machine is just a device with internet connection. It is not directly part of the application but is required to access to functionality. To use the User Webinterface a Google Chrome or Mozilla Firefox Browser is required. The Client does not need a special operating system or system structure to use the project application.

4.2 Front-end

The Front-end provides the User Interface for a simple application access. It is created with HTML5 and JavaScript and redirects the User inputs to API. So the end user must not know the API interface, required data formats and protocols.

4.3 Refining Label API

The Refining Label API is the main access point for the refining algorithm. Every communication goes through this API. After sending data, the API will validate them, execute some preprocessing steps like file conversion and finally execute the refining algorithm and returning the result to the calling user or system. Additionally, the API acts as a connection between the other modules. It connects the user access, algorithm and file handling. Therefore each of the other components is independent and low coupling and high cohesion is achieved.

4.4 File Store

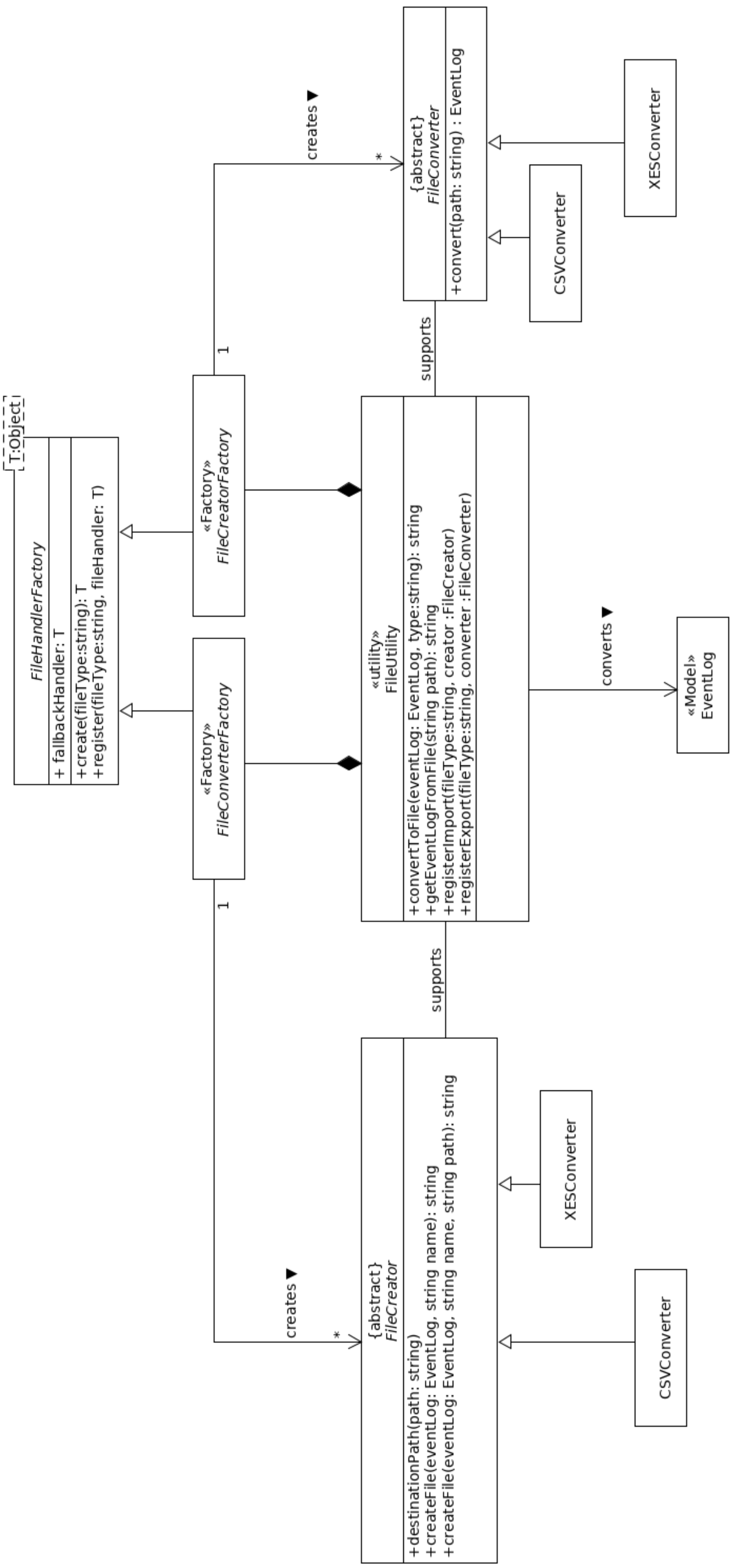
The File Store is an abstraction layer for the servers file system. It will keep track on the updates files and offers functions to store refined XES files on the server. It also monitors the memory utilization.

4.5 Refining Event Labels

The Refining Event Label components is responsible for the final refining algorithm. It contains functionality to calculate costs, execute preprocessing steps, horizontal clustering of variants and the vertical refinement. To offer a maintainable and extensible system this component will provide interfaces to exchange mentioned functionalities.

4.6 Event Log Converter

The Event Log Converter is a Framework to convert different file types into an Event Log and convert them back into files. For our project the system provides functionality for XES and CSV import and export. The Framework can easily be extended with additional file types without changing the core functionality. The following UML class diagram displays the Framework structure:


T:Object
FileHandlerFactory
+ fallbackHandler: T
+create(fileType:string): T
+register(fileType:string, fileHandler: T)
«Factory»
FileCreatorFactory
«Factory»
FileConverterFactory
1
1
creates ▶
*
creates ▶
*
{abstract}
FileConverter
+convert(path: string) : EventLog
CSVConverter
XESConverter
supports
«utility»
FileUtility
+convertToFile(eventLog: EventLog, type:string): string
+getEventLogFromFile(string path): string
+registerImport(fileType:string, creator :FileCreator)
+registerExport(fileType:string, converter :FileConverter)
converts ▶
«Model»
EventLog
supports
{abstract}
FileCreator
+destinationPath(path: string)
+createFile(eventLog: EventLog, string name): string
+createFile(eventLog: EventLog, string name, string path): string
XESConverter
CSVConverter

The FileUtility provides access to the framework and its functionality. The functions *create_file(eventLog: EventLog, type:string): string* realises the XES and CSV file uploader requirement and the *get_event_log_from_file(string path): string* realises the Download refined XES event log file requirement. The Factory and Abstract Class Pattern provides the conversion of multiple file types with the same method. The Factory get the required file type as string and will return the corresponding Converter or Creator. Each file converter must implement the abstract class FileConverter and a file creator must implement the abstract class FileCreator.

If other file types should be supported by the application the developer only has to write another Creator/Converter for the type. Via the *registerImport(fileType:string, creator :FileCreator)* and *registerExport(fileType:string, converter :FileConverter)* methods these classes can be injected in the FileUtility class. Afterwards the functionality is accessible in the whole application, additional code is not required.

5 Policies and Tactics

5.1 Ensuring Requirements Realization

To ensure the realization of all requirements specified in the Requirement Specification Document, the corresponding modules have to be designed. The table below describes which modules are responsible for which requirement and which functionalities are used for the implementation of the module.

Requirements	ID	Modules	Functionalities
-Upload CSV event log file. -Upload XES event log file.	5 6	1.1 File converter. 1.2 Preprocessing log.	1.1.1 getEventLogFromFile() 1.1.2 readXES() 1.2.1 checkEventLog() 1.2.2 lookUpTable() 1.2.3 getVariants()
-Customize threshold for cost function. -Customize horizontal threshold. -Customize vertical threshold. -Choose set of imprecise labels.	7 8 9 10	2. Customize.	2.1 setCandidates() 2.2 getCandidates() 2.3 setVerticalThreshold() 2.4 getVerticalThreshold() 2.5 setHorizontalThreshold() 2.6 getHorizontalThreshold()
-Calculate costs	1	3. Cost function.	3.1 mappings() 3.2 costMapping() 3.3 optimalMatching() 3.4 updateGraph()
-Clustering of traces -Refine labels horizontally across traces.	2 3	4. Horizontal clustering of variants.	4.1 clusterDetection() 4.2 horizontalRefinement()
-Refine labels vertically within traces.	4	5. Vertical refinement.	5.1 verticalRefinement()
−−		6. Post-Processing.	6.1 updateLookUpTable() 6.2 updateEventLog()
-Download refined XES event log file.	11	7. File Creator.	6.1 createFile() 6.2 storeFile()

A detailed description of the modules and functionalities is given in Section 6.

6 Detailed System Design

The main algorithm “Refining Event Labels” will be split up into multiple modules that contain the main parts of the algorithm according to chapter 5.1. These modules will be explained in detail in the following subsections.

6.1 Module 1.1: File Converter

Name: File Creator

Related Component: Event Log Converter.
Type: module.
Description: This module is responsible for creating a table from the data the user uploads. The event log provided by the user is read and stored internally containing all the original columns.
Attributes: none.
Resources: none.
Operations:

1.1.1
Name: getEventLogFromFile()
Arguments: path to a log file in CSV format or XES format.
Returns: event log in XES format.
Description: the file from the path which the user submits is read and stored in XES format. In case the file is originally in CSV format, the file is first converted to XES format and then stored.
Precondition: the path provided by the user leads to a CSV file or an XES file.
Postcondition: the table is stored internally as an XES file.
Exceptions: none.

1.1.2
Name: readXES()
Arguments: a file in XES format.
Returns: an event log object.
Description: the XES file is read and stored as an event log object.
Precondition: an XES file was created using getEventLogFromFile().
Postcondition: an event log object is created and stored internally.
Exceptions: none.

We use the xes_import_factory, i.e. located in pm4py.objects.log.importer.xes.factory file to import a log in xes format.

6.2 Module 1.2: Preprocessing Log

Name: Preprocessing Log.
Related Component: Refining Event Labels.
Type: module.
Description: this module is responsible for preprocessing the data. The file provided by the user is verified, that is, if it has the right standard format, otherwise an error is thrown. Moreover, a table containing all unique traces (i.e. variants) and the list of IDs corresponding to these traces is created.
Attributes: none.
Resources: none.
Operations:

1.2.1
Name: checkEventLog()
Arguments: an event log object obtained from readXES().
Returns: Boolean (True or False).
Description: verify it the event log object has the standard format, i.e., whether it contains an activity column, an ID column and a time stamp column. If these attributes are present in the object, “True” will be returned, otherwise “False” will be returned.
Precondition: an event log object was created using readXES().
Postcondition: if “True” is returned, the event log object is contains all the required attributes, otherwise an error will be displayed.
Exceptions: none.

1.2.2
Name: lookupTable()
Arguments: an event log object.
Returns: lookup table containing a “variants” column and an “ID” column.

Description: the variants are extracted from the event log object and serve as key in the lookup table. The values for each key in the lookup table correspond to the IDs of the traces that belong to a particular variant. The variants are stored as arrays and the corresponding IDs as a list.
Precondition: an already verified event log object.
Postcondition: a lookup table is stored containing the variants and IDs corresponding to those variants.
Exceptions: none.

1.2.3
Name: getVariants()
Arguments: a lookup table obtained from lookupTable().
Returns: set of unique trace variants.
Description: a set containing all the variants is created for further computations (mappings). Here, it is not mandatory to have directly access to the IDs.
Precondition: a non empty lookup table.
Postcondition: a non empty set containing the arrays describing each variant.
Exceptions: none.

Here we can first save all traces in a dictionary data structure where the keys correspond to the case IDs and then use this to extract a list of the case IDs having identical traces.

6.3 Module 2: Customize

Name: Customize.
Related Component: Refining Event Labels.
Type: module.
Description: This module is responsible for customizing the algorithm parameters. The module serves as a model class and it is passed as an argument to the refining algorithm. It stores information about the candidate labels to be refined and the vertical and horizontal threshold. It also automatically validates these parameters.
Attributes: none.
Resources: none.
Operations:

2.1
Name: setCandidates()
Arguments: traces which contain some activity from the candidate set.
Returns: void.
Description: Store the labels which are ”imprecise” according to the user expertise.
Precondition: valid trace model.
Postcondition: traces are stored in the model.
Exceptions: none.

2.2
Name: getCandidates()
Arguments: none.
Returns: a list of event labels.
Description: obtain the labels which are ”imprecise” according to the user expertise.
Precondition: none.
Postcondition: none.
Exceptions: none.

2.3
Name: setVerticalThreshold()
Arguments: vertical threshold as floating point value.
Returns: void.
Description: store the threshold for the vertical refinement. Values bigger than 1 are converted to 0.99 and values smaller than 0 are converted to 0.1
Precondition: none.
Postcondition: none.
Exceptions: none.

2.4
Name: getVerticalThreshold()
Arguments: none.
Returns: vertical threshold as floating point value.
Description: obtain the vertical threshold.
Precondition: none.
Postcondition: none.
Exceptions: none.

2.5
Name: setHorizontalThreshold()
Arguments: horizontal threshold as floating point value.
Returns: void.
Description: store the threshold for the horizontal refinement. Values bigger than 1 are converted to 0.99 and values smaller than 0 are converted to 0.1
Precondition: none.
Postcondition: none.
Exceptions: none.

2.6
Name: getHorizontalThreshold()
Arguments: none.
Returns: horizontal threshold as floating point value.
Description: obtain the horizontal threshold.
Precondition: none.
Postcondition: none.
Exceptions: none.

6.4 Module 3: Cost function

Name: Cost function.
Related Component: Refining Event Labels.
Type: module.
Description: this module is responsible for calculating the costs of all mappings between each pair of variants and selecting the optimal mapping with the least costs.
Attributes: none.
Resources: the weights used for the calculation of costs.
Operations:

3.1
Name: mappings()
Arguments: two distinct trace variants.
Returns: a set of possible mappings.
Description: for each pair of variants the set of common activity labels occurring in both variants is obtained. If this set is empty, no mapping is possible. Otherwise, if none of the common labels appears more than once in any of the variants, the unique mapping is yielded. In the other case, all combinations of possible mappings are yielded as a set where each mapping is an array of pairs of the positions that were matched together.
Precondition: iterate over all pairs of trace variants obtained from getVariants().
Postcondition: for each mapping, the cost function is computed.
Exceptions: none.

3.2
Name: costMapping()
Arguments: two trace variants and a mapping between them.
Returns: the cost of the mapping as a floating point value.
Description: For each pair in the mapping, the number of distinct predecessors and successors and the distances to other matched pairs is counted. Then, a final sum is computed by considering these costs for all matched pairs and also the number of unmatched labels appearing in both traces. Note

that, each term is adjusted with a corresponding weight. Simultaneously all costs between pairs of variants are stored in a list so that the minimal one can be chosen later.
Precondition: all possible mappings are yielded by mappings().
Postcondition: for each mapping, the corresponding cost is computed.
Exceptions: none.

3.3
Name: optimalMapping()
Arguments: the cost of the optimal mapping for each pair of variants.
Returns: a normalized (range [0, 1]) floating point value
Description: The returned value reflects the costs for the best mapping between the pair of variants.
Precondition: the cost of the optimal mapping between two variants is obtained by choosing the minimal entry in the list of costs of their possible mappings calculated in costMapping().
Postcondition: for each pair of variants, the costs of all mappings are known.
Exceptions: none.

3.4
Name: updateGraph()
Arguments: the cost of the optimal mapping for each pair of variants and the set of variants.
Returns: an undirected weighted graph.
Description: For each variant there is a set of vertices corresponding to the events occurring in the variant. Each edge only connects matched pairs and for the pairs being in the candidate set the weight of each edge corresponds to the cost of the optimal mapping. Otherwise the weight of the edge is 0.
Precondition: the weights for the edges are obtained from the optimalMapping().
Postcondition: each variant has to be identified in the graph in order to perform the horizontal and vertical refinements in the next steps.
Exceptions: none.

For the implementation of the graph structure, we use the Python package NetworkX which enables the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

6.5 Module 4: Horizontal clustering of variants

Name: Label refinement based on clusters.
Related Component: Refining Event Labels.
Type: module.
Description: This module clusters the trace variants of the event log and performs label refinement based on these clusters.
Attributes:

- Event log graph which should be refined.
- Customize object for the horizontal thresholds.

Resources: none.
Operations:

4.1
Name: clusterDetection()
Arguments: a graph and a customization Object (Module 2).
Returns: a list of subgraphs.
Description: Group events in the same variant based on their similarity. Two events are considered to be in the same variant if their normalized cost is below the variant threshold z_v. This similarity is expressed by removing edges which cost is below z_v. As a result of removing these edges, a set of subgraphs is obtained and serve as cluster for the horizontal and vertical refinement.
Precondition: a complete undirected weighted graph generated from updateGraph().
Postcondition: a list of undirected weighted subgraphs obtained from updateGraph().
Exceptions: none.

4.2

Name: horizontalRefinement()
Arguments: a list of undirected weighted subgraphs and a customization Object (Module 2).
Returns: void.
Description: Given the set of candidate labels, perform relabelling of events for each cluster of subgraphs, that is, give a new label to each of the events in the set of candidate labels. The relabelling is done exclusively in the graph.
Precondition: a list of undirected weighted subgraphs obtained from clusterDetection().
Postcondition: an updated list of undirected weighted subgraphs with new labels (horizontal refinement).
Exceptions: none.

6.6 Module 5: Vertical Refinement

Name: Label refinement within variant.
Related Component: Refining Event Labels.
Type: module.
Description: This module executes the label refinement within variants for the event log and is part of the refinement algorithm.
Attributes: none.
Resources: none.
Operations:

5.1
Name: verticalRefinement()
Arguments: a list of undirected weighted subgraphs and a customization Object (Module 2).
Returns: void.
Description: given the set of candidate labels and the unfolding threshold v_f, perform relabelling of events as indicated in the section 5.4 from [1].
Precondition: a list of undirected weighted subgraphs with horizontal refinement.
Postcondition: an updated list of undirected weighted subgraphs with new labels (vertical refinement).
Exceptions: none.

6.7 Module 6: Post-Processing

Name: Embed refinements into original log.
Related Component: File converter.
Type: module.
Description: this module is responsible for executing the refinements in the log level. That is, update the original event log with refined labels.
Attributes: the undirected weighted graph containing the new labels for each variant.
Resources: the initial table describing the original event log.
Operations:

6.1
Name: updateLookUpTable()
Arguments: an undirected weighted graph and the lookup table.
Returns: an updated lookup table with refined labels.
Description: replace each old variant in the lookup table with the new refined one.
Precondition: unique case IDs.
Postcondition: an updated lookup table.
Exceptions: none.

6.2
Name: updateEventLog()
Arguments: an updated lookup table.
Returns: an event log object with refined labels for the complete event log.
Description: replace each old variant in the original event log object with the new refined one.
Precondition: an updated lookup table.

Postcondition: the updated event log object contains all original information from the event log but with the refined candidate labels.
Exceptions: none.

6.8 Module 7: File Creator

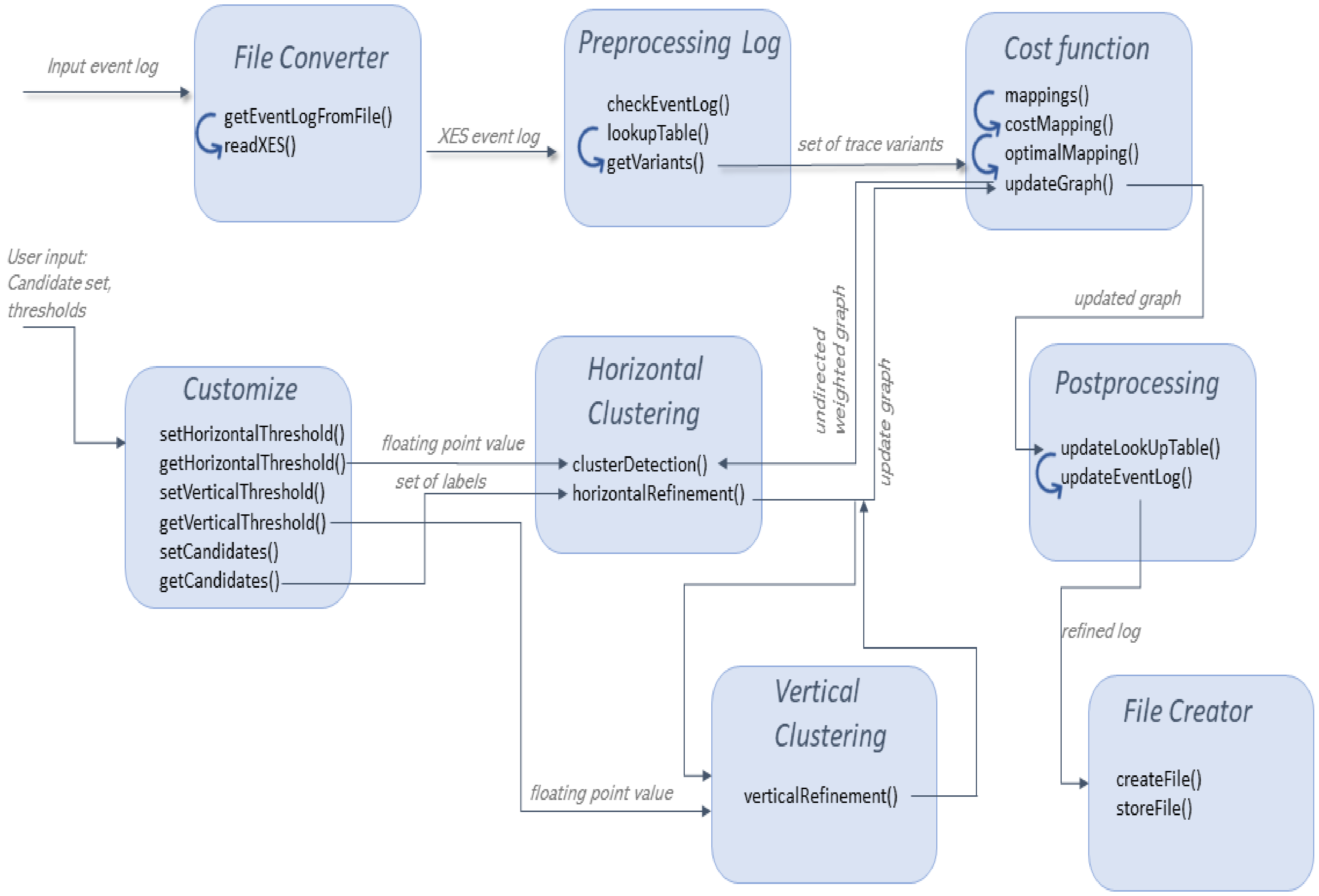
Name: File Creator
Related Component: Refining Event Labels / File Upload.
Type: module.
Description: utility class to export and store an event logs as a XES file.
Attributes: an event log object.
Resources: none.
Operations:

6.1
Name: createFile()
Arguments: event log object.
Returns: void.
Description: converts the event logs a XES file.
Precondition: a valid event log format.
Postcondition: a XES file is created.
Exceptions: none.

6.2
Name: storeFile()
Arguments: a directory path.
Returns: the create file path.
Description: stores the previous created file at given location.
Precondition: an existing directory Path.
Postcondition: the XES file is stored.
Exceptions: NotADirectoryException

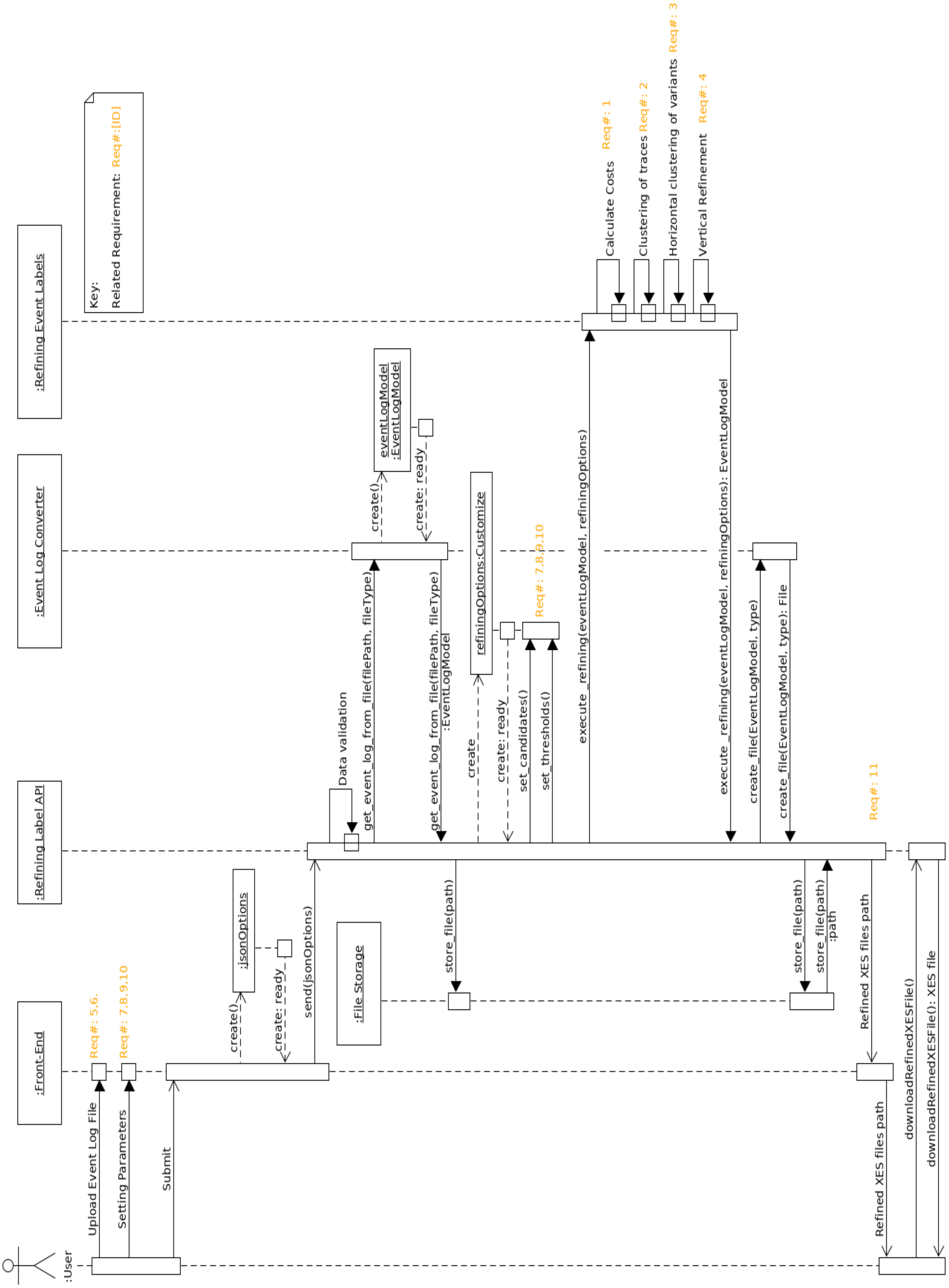
6.9 Data Flow Diagramm

Data Flow Diagramm



6.10 Uses/Interactions

The components from the section System Architecture and the related modules result in the following System flow:


:Refining Event Labels
Key:
Related Requirement: Req#:[ID]
:Event Log Converter
:Refining Label API
:Front-End
:User
Calculate Costs Req#: 1
Clustering of traces Req#: 2
Horizontal clustering of variants Req#: 3
Vertical Refinement Req#: 4
eventLogModel :EventLogModel
create()
create: ready
refiningOptions:Customize
Req#: 7,8,9,10
execute _refining(eventLogModel, refiningOptions)
execute _refining(eventLogModel, refiningOptions): EventLogModel
create_file(EventLogModel, type)
create_file(EventLogModel, type): File
Data validation
get_event_log_from_file(filePath, fileType)
get_event_log_from_file(filePath, fileType) :EventLogModel
create
create: ready
set_candidates()
set_thresholds()
Req#: 11
:jsonOptions
create()
create: ready
send(jsonOptions)
:File Storage
store_file(path)
store_file(path)
store_file(path) :path
Refined XES files path
Req#: 5.6.
Req#: 7,8,9,10
Upload Event Log File
Setting Parameters
Submit
Refined XES files path
downloadRefinedXESFile()
downloadRefinedXESFile(): XES file

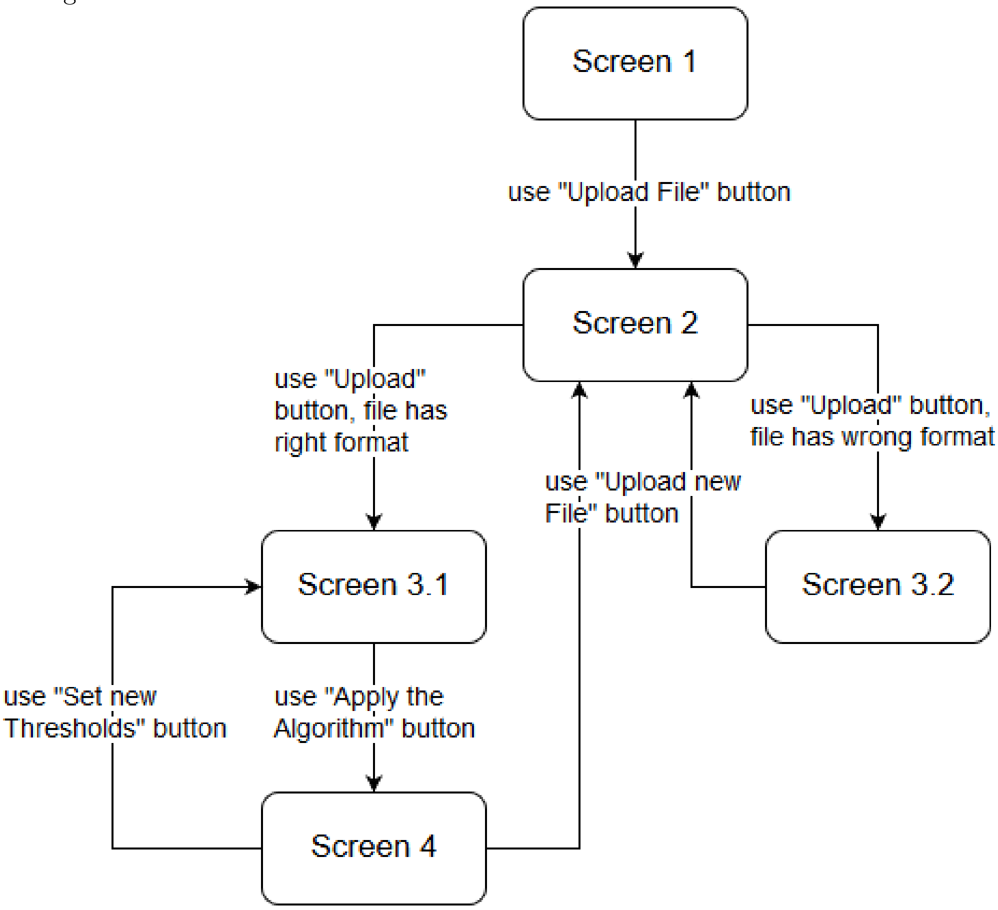
7 User Interface Design

7.1 Application Control

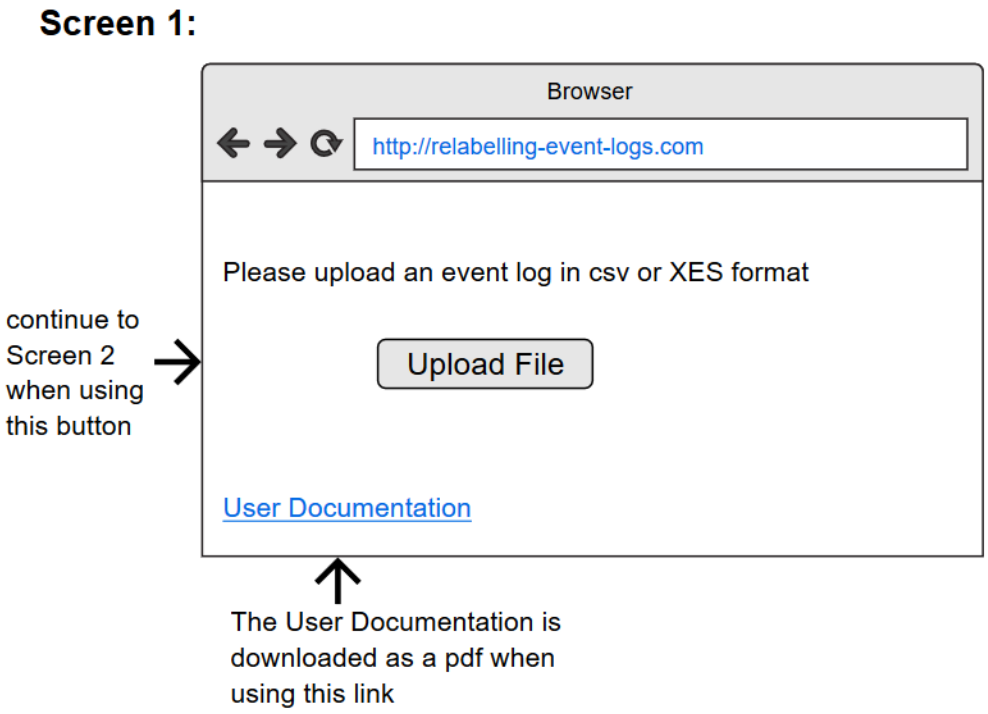
In the project, we will implement a web service. The web client will have a rather plain design that should focus on the main activities the service should provide, which are uploading event logs in CSV or XES format, setting the threshold for the label refinement algorithm and download the refined log after the algorithm is finished. There will be buttons used to upload the file, apply the algorithm and download the refined event log. Moreover, the screens will have short explanations telling the user what to do (if not self-explanatory). For setting the thresholds, two boxes will be provided that include the default thresholds, but new values can be entered by the user. A draft of each of the main screen can be found in the next section, section 6.2.

7.2 The Screens

The main screens will be visualized in the following subsections. In these screens include the main functionalities, which are described in the former section. The following diagram will show the flow of control through the screens.


Screen 1
use "Upload File" button
Screen 2
use "Upload" button, file has right format
use "Upload new File" button
use "Upload" button, file has wrong format
Screen 3.1
Screen 3.2
use "Set new Thresholds" button
use "Apply the Algorithm" button
Screen 4

7.2.1 Screen 1


Screen 1:

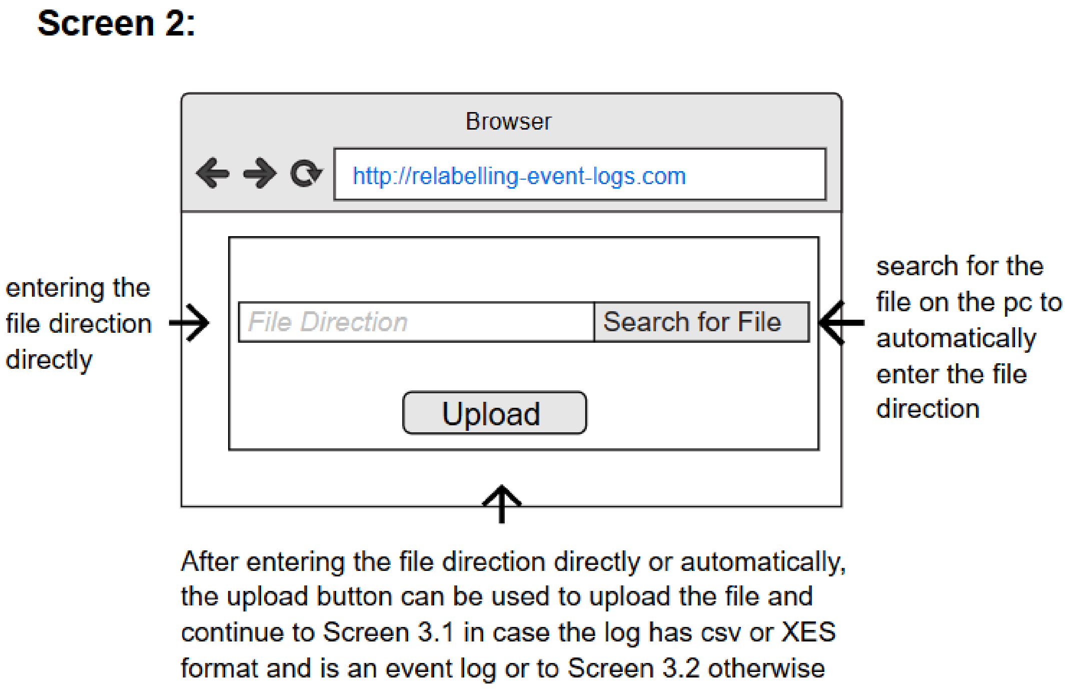
Please upload an event log in csv or XES format

continue to Screen 2 when using this button

The User Documentation is downloaded as a pdf when using this link

The first screen visible to the user will show a description saying that an event log in CSV of XES format should be uploaded. Moreover, a button “Upload File” is visible. By using this button, the user will continue to Screen 2. At the end of the page, there will be a link called “User Documentation”. By clicking on this link, the User Documentation will be downloaded in pdf format.

7.2.2 Screen 2


Screen 2:

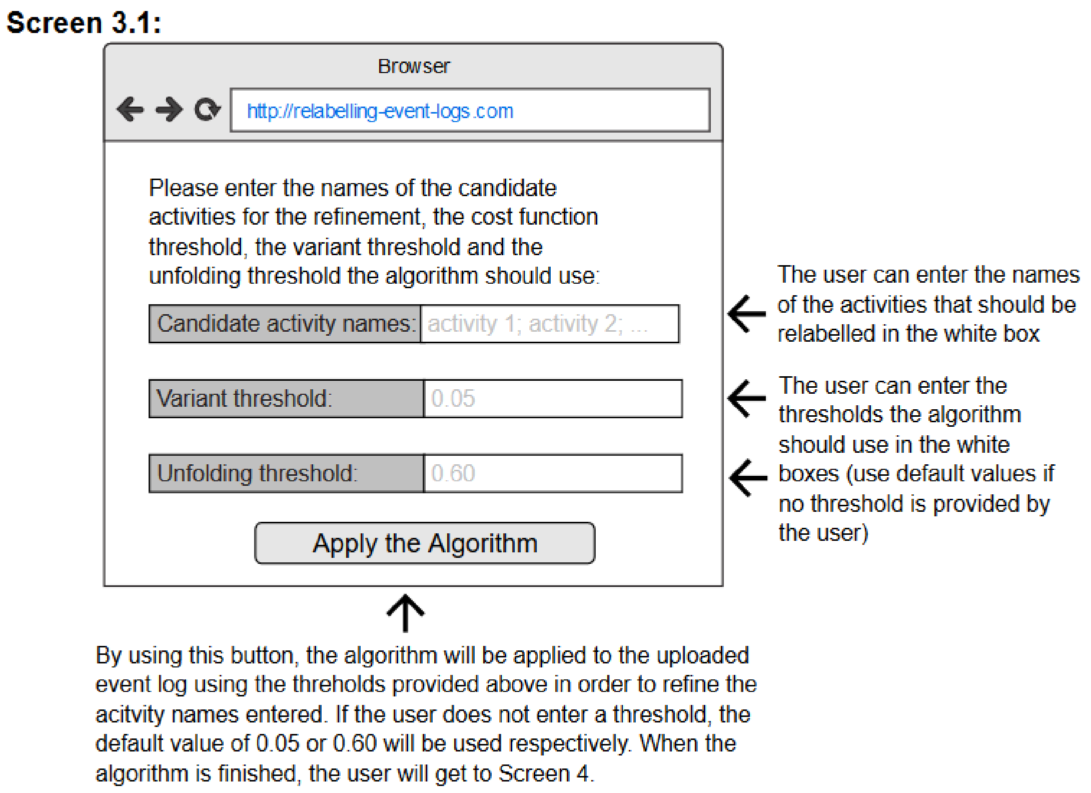
entering the file direction directly

search for the file on the pc to automatically enter the file direction

After entering the file direction directly or automatically, the upload button can be used to upload the file and continue to Screen 3.1 in case the log has csv or XES format and is an event log or to Screen 3.2 otherwise

In the second screen visible to the user, the user can enter the file direction of the event log he wants to upload. He can either directly type in the direction into the “File Direction” field or use the “search for File” button to search for a file on his PC, so that the direction will automatically be filled in after selecting a file. After using one of this alternatives, he can use the “Upload” button to upload the file with the given directory. In case this file is an event log, i.e., the data contains at least the attributes “id”, “time stamp” and “activity name”, and has either CSV or XES format, the user will continue to Screen 3.1. If one of these conditions is not satisfied, he will continue to Screen 3.2.

7.2.3 Screen 3.1



Screen 3.1:

Please enter the names of the candidate activities for the refinement, the cost function threshold, the variant threshold and the unfolding threshold the algorithm should use:

Candidate activity names: activity 1; activity 2; ...

Variant threshold: 0.05

Unfolding threshold: 0.60

The user can enter the names of the activities that should be relabelled in the white box

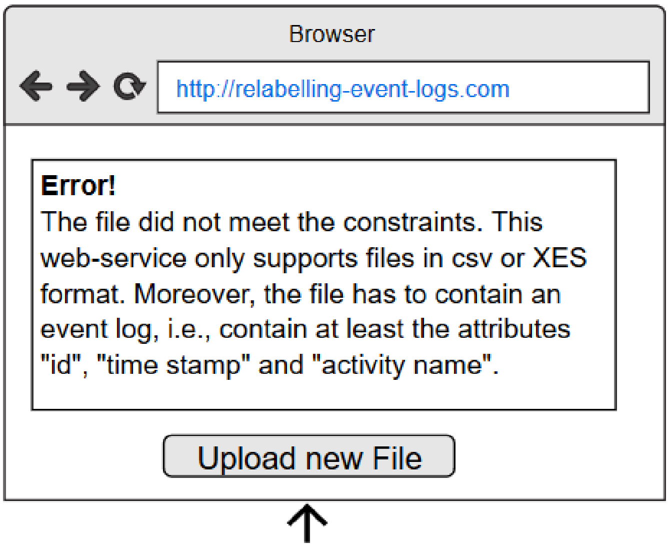
The user can enter the thresholds the algorithm should use in the white boxes (use default values if no threshold is provided by the user)

By using this button, the algorithm will be applied to the uploaded event log using the threholds provided above in order to refine the acitvity names entered. If the user does not enter a threshold, the default value of 0.05 or 0.60 will be used respectively. When the algorithm is finished, the user will get to Screen 4.

This screen appears if the file uploaded by the user meets the requirements. In this screen, the user can set the candidate activity names, i.e., those activities that the algorithm can relabel. Moreover, the user can set the thresholds for the algorithm, i.e., the variant and the unfolding threshold. He can enter all of these in the corresponding white boxes. If he does not enter the thresholds, the default values of 0.05 and 0.60 will be used respectively. Using the button “Apply the Algorithm”, the web service will start applying the algorithm using the provided thresholds. After the algorithm is finished, the user will get to Screen 4.

7.2.4 Screen 3.2

Screen 3.2:



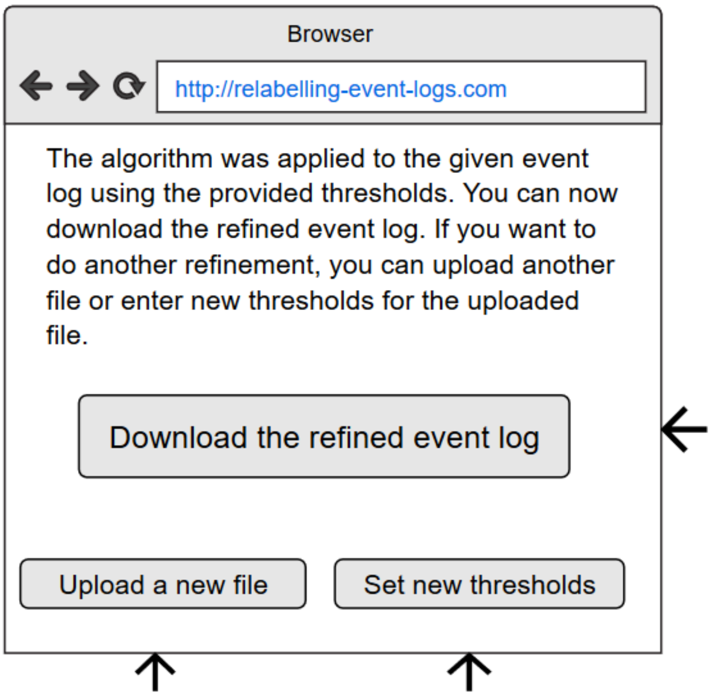
Error!
The file did not meet the constraints. This web-service only supports files in csv or XES format. Moreover, the file has to contain an event log, i.e., contain at least the attributes "id", "time stamp" and "activity name".

By using this button, the user will return to Screen 2.

This screen appears if the upload was not successful because the file did not meet the assumptions. If the file does not have the right format, the user can click on the button “Upload new File” to return to Screen 2 and upload a file that meets the constraints.

7.2.5 Screen 4

Screen 4:


Browser
http://relabelling-event-logs.com
Download the refined event log
Upload a new file
Set new thresholds

The algorithm was applied to the given event log using the provided thresholds. You can now download the refined event log. If you want to do another refinement, you can upload another file or enter new thresholds for the uploaded file.

Using this button, the user can download the refined evend log

By using this button, the user will return to Screen 2.

By using this button, the user will return to Screen 3.1.

This Screen will be shown after finishing the algorithm. The user can now download the refined log using the corresponding button. After this step, the user is done and can exit the page, but if he also wants to apply the algorithm to another event log or to the same event log using different thresholds, he can use the corresponding buttons and will be redirected to Screen 2 or Screen 3.1 respectively.

References

[1] Lu, Xixi, et al. "Handling duplicated tasks in process discovery by refining event labels." International Conference on Business Process Management. Springer, Cham, 2016.

[2] Xixi Lu1, Dirk Fahland, Frank J.H.M. van den Biggelaar, Wil M.P. van der Aalst. "Detecting Deviating Behaviors without Models."