

Software Design Specification

Label Refinement by Behavioral Similarity

Document owners:

Bianka Bakullari
Christopher Beine
Nicole Ventsch
Juan Garza

Last edited: May 11, 2019

1	Introduction	1
1.1	System Overview	1
1.2	Design Map	1
1.3	Supporting Materials	1
1.4	Definitions and Acronyms	1
2	Design Considerations	1
2.1	Assumptions and Dependencies	1
2.2	General Constraints	1
2.3	Goals and Guidelines	1
2.4	Development Methods	1
3	Architectural Strategies	1
4	System Architecture	1
4.1	Client Machine	2
4.2	Front-end	2
4.3	Refining Label API	2
4.4	File Store	2
4.5	Refining Event Labels	2
4.6	Event Log Converter	2
5	Policies and Tactics	4
5.1	Ensuring Requirements Realization	4
6	Detailed System Design	4
6.1	Module 1.1: File Converter	4
6.2	Module 1.2: Preprocessing Log	5
6.3	Module 2: Customize	6
6.4	Module 3: Cost function	7
6.5	Module 4: Horizontal clustering of variants	8
6.6	Module 5: Vertical Refinement	8
6.7	Module 6: Post-Processing	9
6.8	Module 7: File Creator	9
7	User Interface Design	10
7.1	Application Control	10
7.2	The Screens	10
7.2.1	Screen 1	11
7.2.2	Screen 2	11
7.2.3	Screen 3.1	12
7.2.4	Screen 3.2	13

7.2.5 Screen 4 13

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

2.2 General Constraints

2.3 Goals and Guidelines

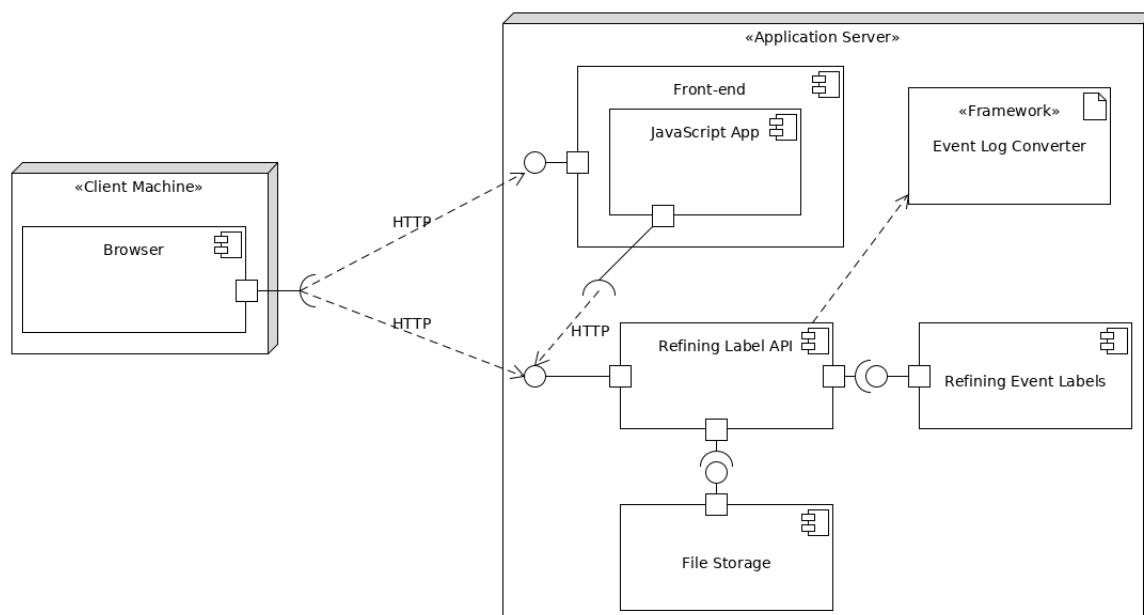
2.4 Development Methods

3 Architectural Strategies

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 servers 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 option to perform the algorithm:

- 1. The user can access the JavaScript Front-end application via HTTP which provides an user friendly UI. There the user can set all required parameters and send them to the API.
- 2. Additional, the user can access the API directly via HTTP with the required parameter. So the application can integrate into other 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 user 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. Additional the API acts as a connection between the other Modules. Is 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

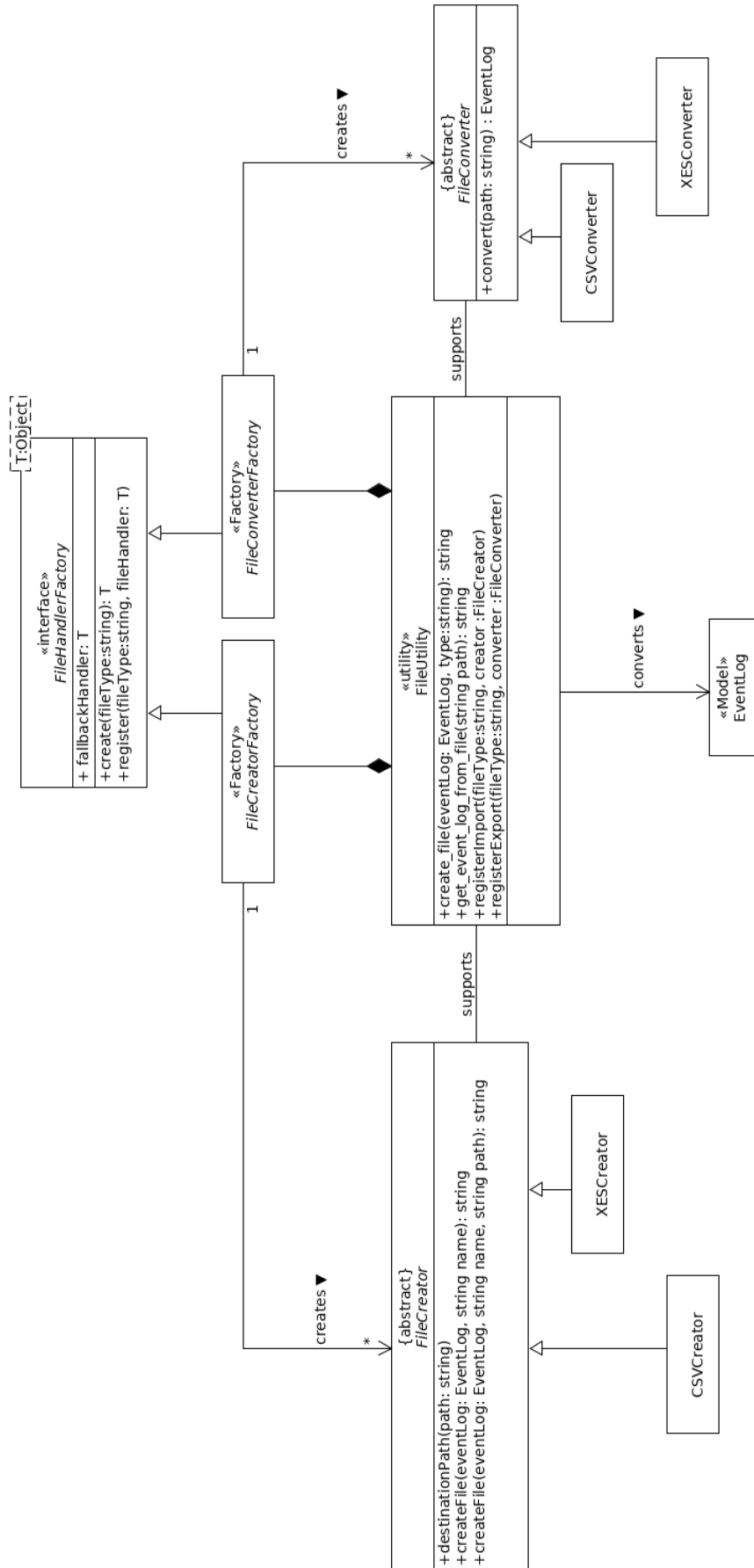
The File Store is an abstraction layer for the servers file system. It will keep track on the updates files and offers functions to store refined XES files on the server. It also monitors the memory utilization.

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 provides interfaces to exchange mentioned functionalities. The following subcomponents exist:

4.6 Event Log Converter

submodules The Event Log Converter is a Framework to convert different files 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 display the Framework structure:



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 methode. 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. Afterward the functinalty is accessible in the whole applicatoin, 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	Modules	Functionalities
-Upload CSV event log file -Upload XES event log file	1.1 File Converter 1.2 Preprocessing log	1.1.1 get_event_from_file() 1.1.2 read_XES() 1.2.1 check_event_log() 1.2.2 lookup_table() 1.2.3 get_variants
-Customize threshold for cost function -Customize horizontal threshold -Customize vertical threshold -Choose set of imprecise labels	2. Customize	2.1 set_candiadates() 2.2 get_candiadates() 2.3 set_threshold() 2.4 get_vertical_threshold() 2.5 get_horizontal_threshold()
-Calculate costs	3. Cost function	3.1 mappings() 3.2 cost_mapping() 3.3 cost_matrix() 3.4 graph()
-Clustering of traces -Refine labels horizontally across traces	4. Horizontal clustering of vari- antes	4.1 cluster_detection_relation() 4.2 refinement()
-Refine labels vertically within traces	5. Vertical refinement	5.1 relabel()
--	6. Post-Processing	6.1 update_map() 6.2 update_table()
-Download refined XES event log file	7. File Creator	6.1 create_file() 6.2 store_file()

A detailed description of the modules and funcnionalities 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. 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: get_event_log_from_file()

Arguments: path to a log file in csv format or XES format

Returns: event log in XES format

Description: The file at the path the user enters 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: read_XES()

Arguments: file in XES format

Returns: event log

Description: The XES file is read and stored as a table containing the original columns.

Precondition: an XES file was created using get_event_log_from_file()

Postcondition: a table containing all original columns is stored internally

Exceptions: None

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. It is checked if the table provided by the user contains the right variables and an error is produced if it does not have. Moreover, a table containing all unique traces and the list of IDs corresponding to these traces is created.

Attributes: None

Resources: None

Operations:

1.2.1

Name: check_event_log()

Arguments: table imported using read_XES()

Returns: Boolean (True or False)

Description: It is checked whether the table is actually an event log, i.e., whether it contains an activity column, an ID column and a time stamp column. If these exist in the table, "True" will be returned, otherwise "False" will be returned.

Precondition: a file was created using read_XES()

Postcondition: if "True" is returned, the file is actually an event log, otherwise the file does not contain the necessary variables and an error will be displayed

Exceptions: None

1.2.2

Name: lookup_table()

Arguments: table with all original columns

Returns: lookup table containing a "variants" column and an "ID" column

Description: The table is converted, so that we get a table containing the unique variants for the IDs ordered by their time stamp and the set of IDs corresponding to the variant. In this lookup table, the variants are stored as arrays and the corresponding IDs as a list.

Precondition: the provided file is an event log

Postcondition: a lookup table is stored containing the variants and IDs corresponding to those variants

Exceptions: None

1.2.3

Name: get_variants()

Arguments: look-up table with variants and IDs for each variant

Returns: set of unique trace variants

Description: We create a set containing all the variants so that we can approach them when calculating the mappings later. Here we do not need to have access to the IDs.

Precondition: The first column of the look-up table contains all trace variants

Postcondition: a non-empty set containing the arrays describing each variant

Exceptions: None

6.3 Module 2: Customize

Name: Customize

Related Component: Refining Event Labels

Type: module

Description: This module is responsible to customize the algorithm parameters. The module serves as a model class and will later pass to the refining algorithm. It stores information about the candidates to refine, and the vertical and horizontal threshold. Also it will automatically validate this data.

Attributes: None

Resources: None

Operations:

2.1

Name: set_candiadates()

Arguments: traces for which should be refined.

Returns: void

Description: The function provides functionality to stores the traces for the final refining algorithm.

Precondition: Valid trace model

Postcondition: traces are stored in the model

Exceptions: None

2.2

Name: get_candiadates()

Arguments: none.

Returns: Array of traces which could be considered by the refining algorithm.

Description: Provides access to all traces

Precondition: none.

Postcondition: none.

Exceptions: None

2.3

Name: set_threshold()

Arguments: vertical and horizontal threshold as double values.

Returns: void

Description: Allows to set the threshold for the horizontal and vertical refinement. Values bigger than 1 will be converted to 0.99 and values smaller than 0 will be converted to 0.1

Precondition: none.

Postcondition: none.

Exceptions: None

2.4

Name: get_vertical_threshold()

Arguments: none.

Returns: vertical threshold as double

Description: function return the vertical threshold

Precondition: none.

Postcondition: none.

Exceptions: None

2.5

Name: get_horizontal_threshold()
Arguments: none.
Returns: horizontal threshold as double
Description: function return the vertical 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 we obtain the set of common activity labels occurring in both variants. 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: We go through all pairs of trace variants yielded by get_variants()
Postcondition: For each mapping, the cost function is computed
Exceptions: None

3.2

Name: cost_mapping()
Arguments: two trace variants and a mapping between them
Returns: the cost of the mapping as a real number
Description: For each pair in the mapping we count the number of distinct predecessors and successors and the distances to other matched pairs. We sum over these costs for all matched pairs and also add the number of unmatched labels appearing in the traces. Each summand is weighted with a corresponding weight. Simultaneously we save all costs between pairs of variants in a list so that we can pick the minimal one.
Precondition: We calculate the costs for each mapping yielded by mappings().
Postcondition: For each mapping, the corresponding cost is computed.
Exceptions: None

3.3

Name: cost_matrix()
Arguments: the cost of the optimal mapping for each pair of variants
Returns: a symmetrical 2-dimensional matrix
Description: The matrix contains 0s in the diagonal and the entry in position $[i][j]$ corresponds to the cost of the optimal mapping between variant i and variant j .
Precondition: We can obtain the cost of the optimal mapping between two variants by choosing the minimal entry in the list of costs of their possible mappings saved in cost_mapping().
Postcondition: For each pair of variants, the cost of the optimal mapping is known.
Exceptions: None

3.4

Name: graph()
Arguments: the cost of the optimal mapping for each pair of variants and the set of variants

Returns: a connected 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 `cost_matrix()`.

Postcondition: Each variants has to be identifiable in the graph in order to be able to do the horizontal and vertical refinements in the next steps.

Exceptions: None

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 event log and perfoms the 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: `cluster_detection_relation()`

Arguments:

Returns: void.

Description: Finds clusters inside.

Precondition: An event log graph with related costs.

Postcondition: The same graph object with detected clusters.

Exceptions: None

4.2

Name: `refinement()`

Arguments: none.

Returns: A horizontal clustered graph.

Description: This method executes required steps to cluster a graph horizontal.

Precondition: A already clustered graph.

Postcondition: The same graph object with horizontal refined labels.

Exceptions: None

Operations: `clusterdetectionrelation`, `refinement`

6.6 Module 5: Vertical Refinement

Name: Label refinement within variant

Related Component: Refining Event Labels

Type: module

Description: This module execute the label refinement within variant for the Event log and is part of the refinement algorithm.

Attributes: none.

Resources: none.

Operations:

5.1

Name: `relabel()`

Arguments: a graph, the associated cluster and the customization Object for the horizontal threshold,

Returns: The realbeld graph

Description: Performs the vertical label refinement for the refinement algorithm.

Precondition: An already horizontal refined graph.

Postcondition: The same graph object with vertical refined labels.

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, we want to go recreate the original event log with refined labels.

Attributes: The graph containing the new labels for each variant

Resources: The initial table describing the original event log

Operations:

6.1

Name: update_map()

Arguments: a graph and the look-up map

Returns: the updated look-up with refined labels

Description: We replace each old variant in the look-up map with the new refined one.

Precondition: We have to be able to relate each old variant to its corresponding refined version.

Postcondition: For each refined variant, we still have the list of the corresponding case IDs.

Exceptions: None

6.2

Name: update_table()

Arguments: the updated look-up map

Returns: the updated table with refined labels for the complete log

Description: We replace each old variant in the initial table map with the new refined one.

Precondition: We use the case IDs to be able to substitute the old variants with the new ones.

Postcondition: The updated table contains all original information from the event log but with refined candidate labels.

Exceptions: None

6.8 Module 7: File Creator

Name: File Creator

Related Component: Refining Event Labels

Type: module

Description: Utility class to export and store an Event Logs as a XES file.

Attributes: An Event Log model

Resources: none.

Operations:

6.1

Name: create_file()

Arguments: Event Log model.

Returns: void.

Description: Converts the event logs a XSD file.

Precondition: A valid event log format.

Postcondition: a XSD file is created.

Exceptions: None

6.2

Name: store_file()

Arguments: A directory path.

Returns: the create file path.

Description: Stores the previous created file at given location.

Precondition: A existing directory Path.

Postcondition: the XSD file is stored.

Exceptions: NotADirectoryException

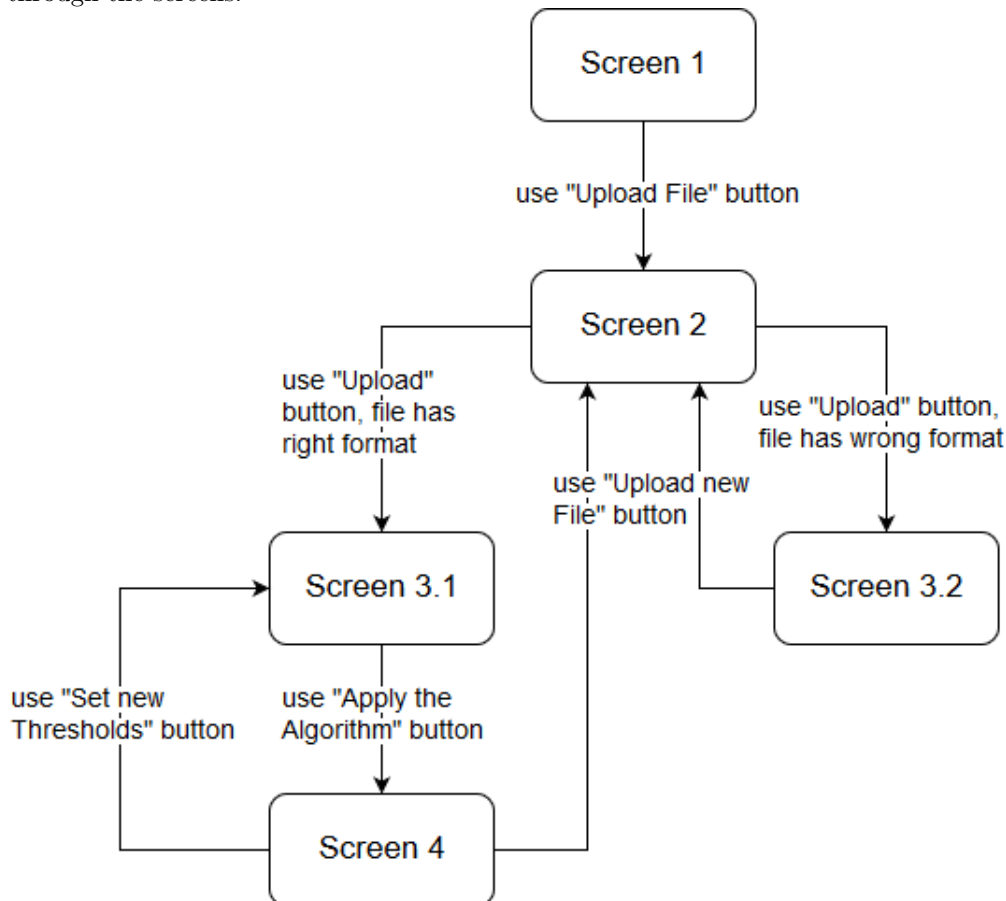
7 User Interface Design

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.

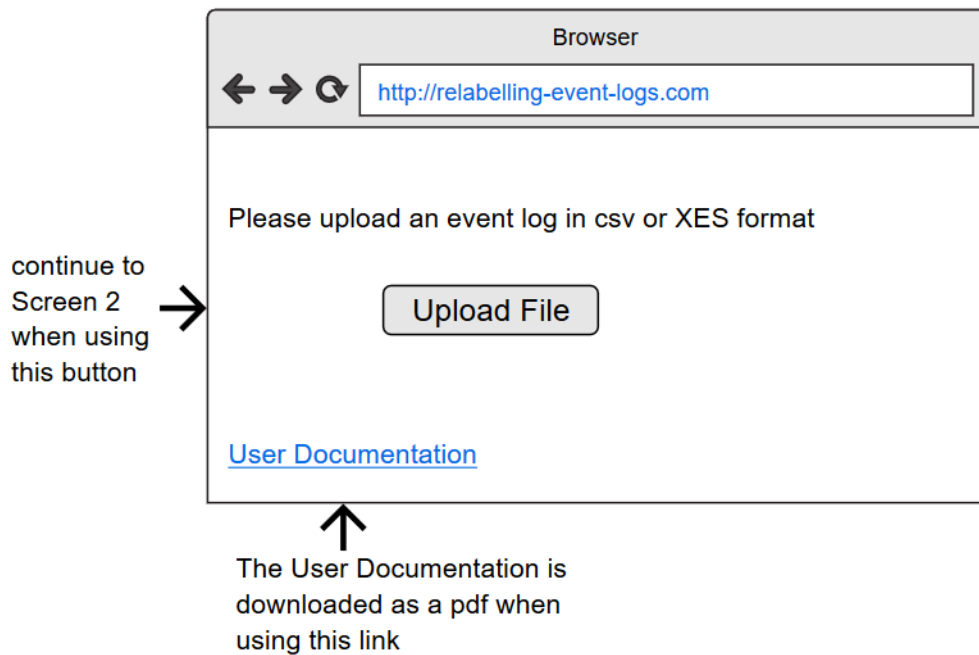
7.2 The Screens

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.



7.2.1 Screen 1

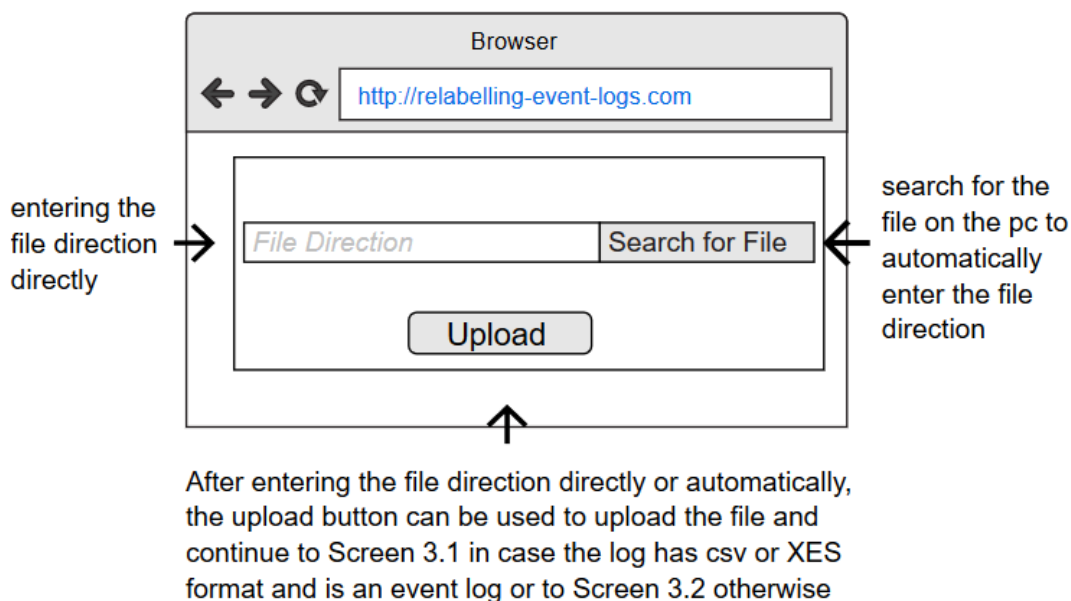
Screen 1:



The first screen visible to the user will show a description saying that an event log in csv or 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:



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:

Browser

← → ↻

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:

Variant threshold:

Unfolding threshold:

← 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)

← 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)

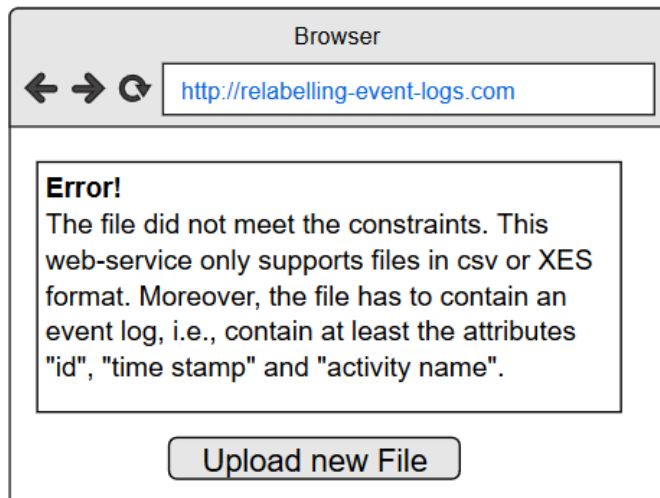
↑

By using this button, the algorithm will be applied to the uploaded event log using the thresholds provided above in order to refine the activity 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 thresholds for the algorithm, i.e., the variant and the unfolding threshold. He can enter 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:

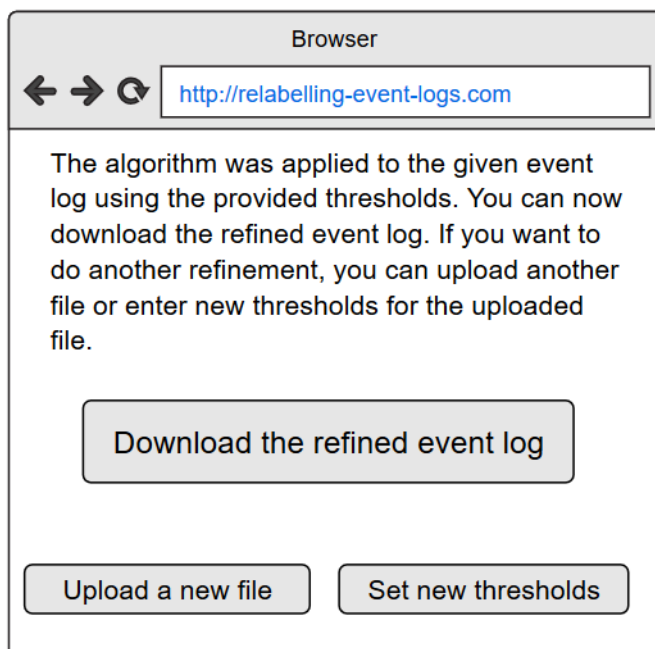


By using this button, the user will return to Screen 2.

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:



Using this button, the user can download the refined event 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."