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

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 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 option to perform the algorithm:

- 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.

- 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. .
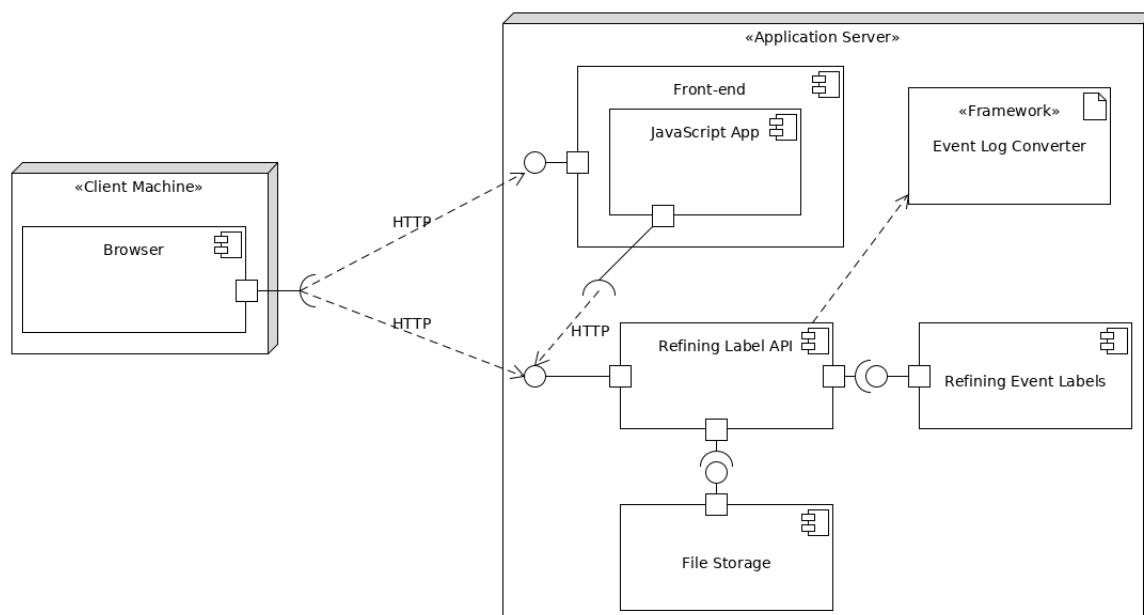
4.1 Client Machine

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

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:

FileHandlerFactory [T:Object]
- fallbackHandler: T
+ create(fileType:string): T
+ register(fileType:string, fileHandler: T)

«Factory» **FileCreatorFactory**

«Factory» **FileConverterFactory**

«utility» **FileUtility**
+ convertToFile(eventLog: EventLog, type:string): string
+ getEventLogFromFile(string path): string
+ registerImport(fileType:string, creator :FileCreator)
+ registerExport(fileType:string, converter :FileConverter)

«Model» **EventLog**

{abstract} *FileConverter*
+ convert(path: string) : EventLog

CSVConverter

XESConverter

{abstract} *FileCreator*
+ destinationPath(path: string)
+ createFile(eventLog: EventLog, string name): string
+ createFile(eventLog: EventLog, string name, string path): string

XESConverter

CSVConverter

creates ▶

supports

converts ▶

supports

1

*

1

*

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 createlookUpTable() 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 - Probably is a good idea to have an optimalMatching() function that gets as a parameters the set of possible mappings (i.e. from the mappings() function) and returns an object with the best mapping and its cost. Then we project that cost directly to the edges of the graph instead of storing them in the symmetric matrix (i.e. costMatrix()) in this way we save memory space.	1	3. Cost function.	3.1 mappings() 3.2 costMapping() 3.3 costMatrix() 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.

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.

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 it is passed as an argument to the refining algorithm. It stores information about the candidates to refine and the vertical and horizontal threshold. It also automatically validates these parameters.
Attributes: none.
Resources: none.
Operations:

2.1
Name: setCandidates()
Arguments: traces for which should be refined.
Returns: void.
Description: store the traces 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 traces.
Description: obtain the traces 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 is chosen.

Precondition:calculate the costs for each mapping yielded by mappings().

Postcondition: for each mapping, the corresponding cost is computed.

Exceptions: none.

3.3

Name: costMatrix()

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: 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 stored in costMapping().

Postcondition: for each pair of variants, the cost of the optimal mapping is 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 costMatrix().

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.

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 performs 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: 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 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: 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() Here, what is table referred to? The event log? I changed to updateEventLog() since we are using an event log object but feel free to make any changes :)
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.
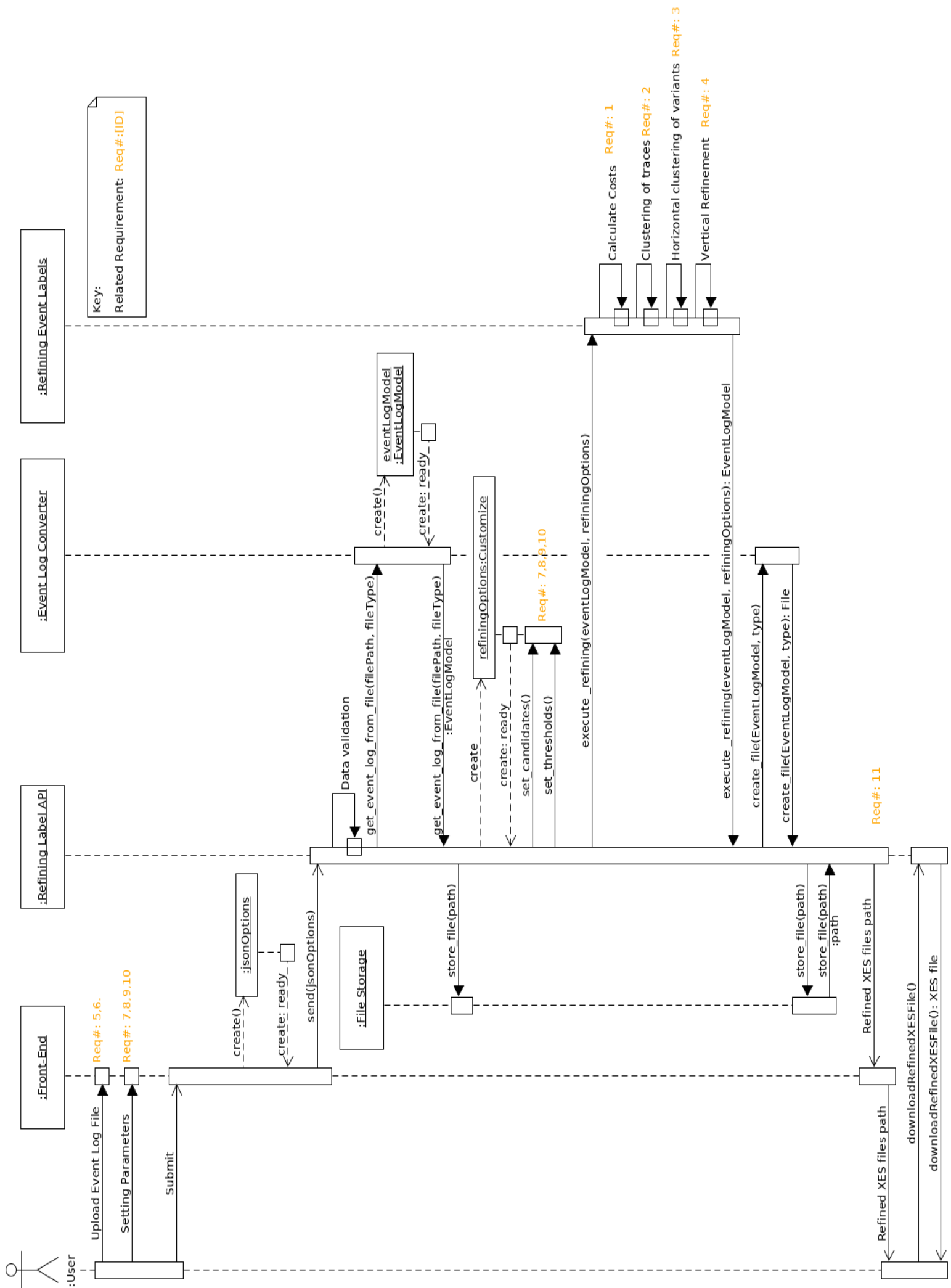*Description*: stores the previous created file at given location.
Precondition: a existing directory Path.
Postcondition: the XES file is stored.
Exceptions: NotADirectoryException

6.9   Uses/Interactions

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

Sequence diagram: Refining Event Labels

Participants: :User, :Front-End, :Refining Label API, :File Storage, :Event Log Converter, :Refining Event Labels

Key:
Related Requirement: Req#: [ID]

Req#: 5,6.
Req#: 7,8,9,10

User → Front-End: Upload Event Log File
User → Front-End: Setting Parameters
User → Front-End: Submit

Front-End: create() :jsonOptions
:jsonOptions create: ready
Front-End → Refining Label API: send(jsonOptions)

Refining Label API: Data validation

Refining Label API → Event Log Converter: get_event_log_from_file(filePath, fileType)
Event Log Converter: create() eventLogModel :EventLogModel
eventLogModel create: ready
get_event_log_from_file(filePath, fileType): EventLogModel

Refining Label API: create refiningOptions:Customize
refiningOptions create: ready
Req#: 7,8,9,10
set_candidates()
set_thresholds()

Refining Label API → Refining Event Labels: execute_refining(eventLogModel, refiningOptions)

Refining Event Labels:
- Calculate Costs   Req#: 1
- Clustering of traces   Req#: 2
- Horizontal clustering of variants   Req#: 3
- Vertical Refinement   Req#: 4

execute_refining(eventLogModel, refiningOptions): EventLogModel

Refining Label API → Event Log Converter: create_file(EventLogModel, type)
create_file(EventLogModel, type): File

Refining Label API → File Storage: store_file(path)
File Storage → : store_file(path) :path
store_file(path):path

Req#: 11

Refining Label API → Front-End: Refined XES files path
Front-End → User: Refined XES files path

User → Front-End: downloadRefinedXESFile()
Front-End → Refining Label API: downloadRefinedXESFile(): XES file

:File Storage
store_file(path)

11

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.
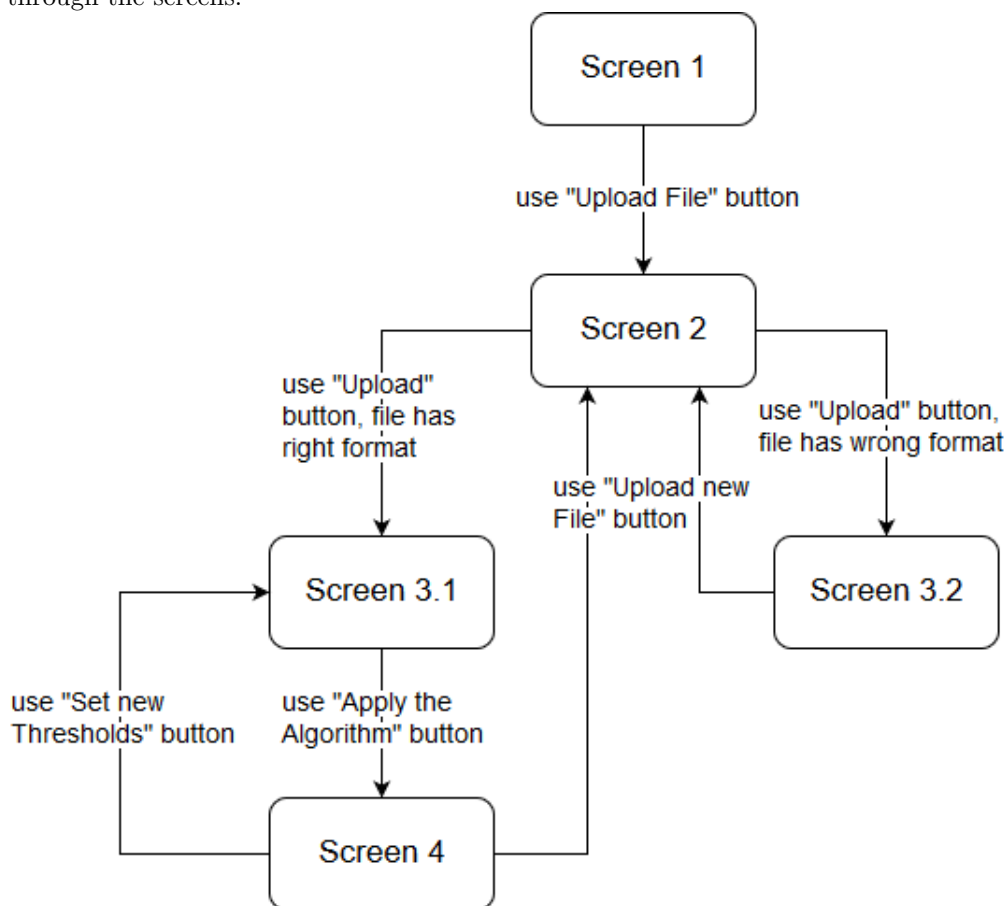
7.2.1 Screen 1

Screen 1:
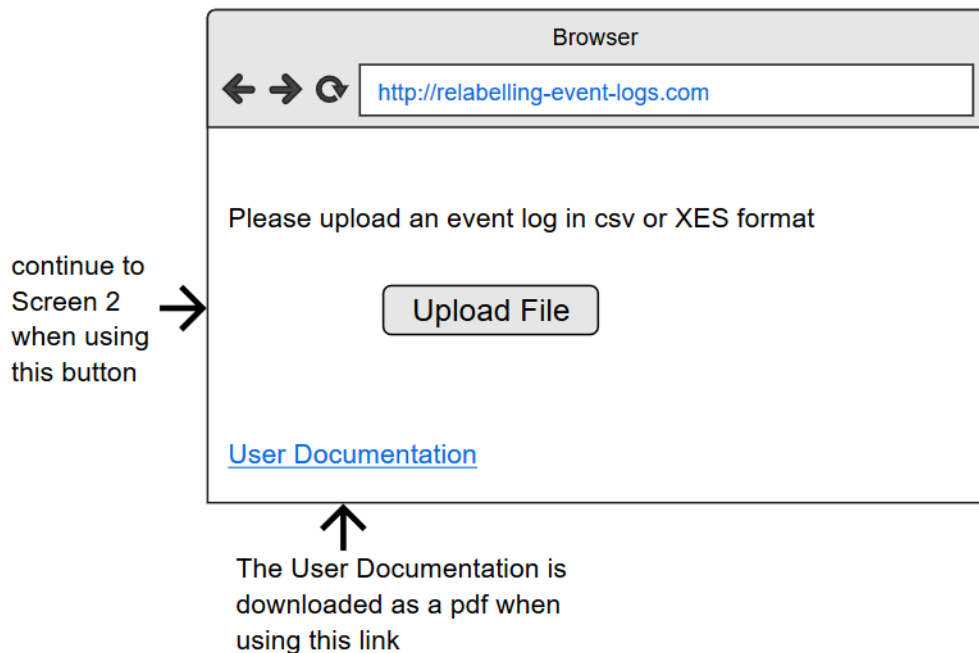
The first screen visible to the user will show a description saying that an event log in CSV of XES format should be uploaded. Moreover, a button "Upload File" is visible. By using this button, the user will continue to Screen 2. At the end of the page, there will be a link called "User Documentation". By clicking on this link, the User Documentation will be downloaded in pdf format.

7.2.2 Screen 2

Screen 2:
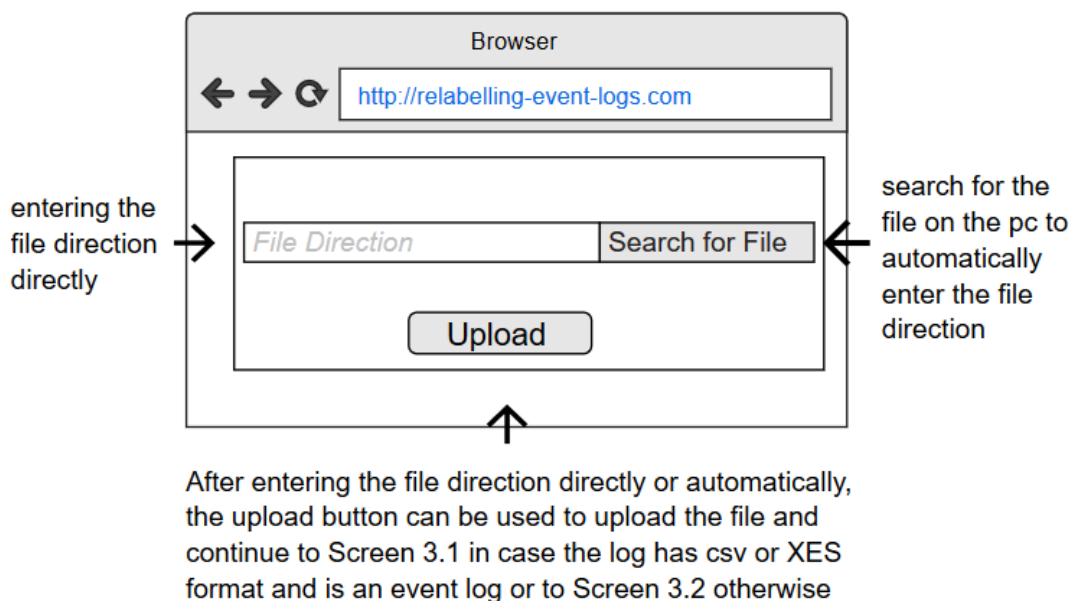
In the second screen visible to the user, the user can enter the file direction of the event log he wants to

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

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.
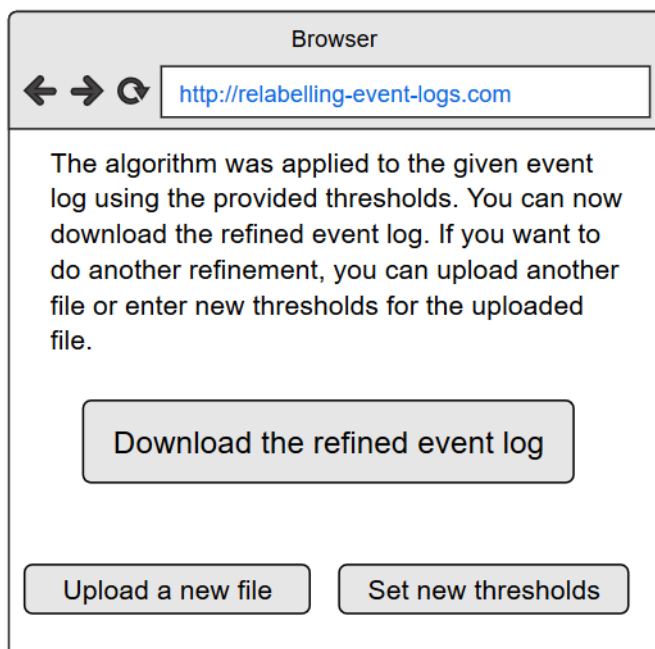
7.2.4 Screen 3.2

Screen 3.2:
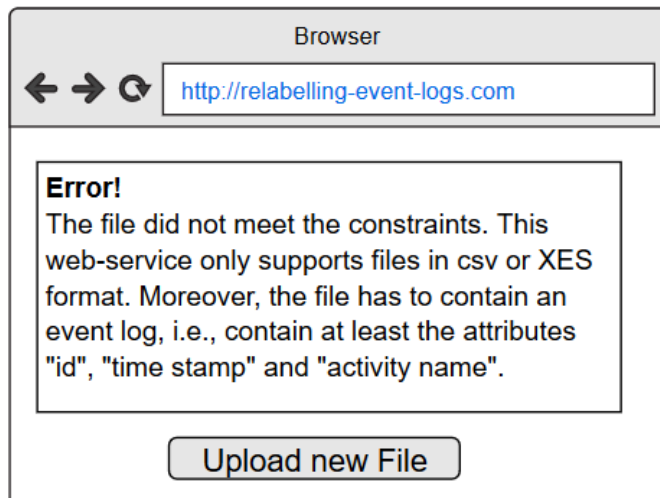
By using this button, the user will return to Screen 2.

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 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."