Matija Striga

Automatic Evaluation of User Experiments using Intermediate Language and Compiler

Bachelor Thesis

Graz University of Technology

Institute for Softwaretechnology
Head: Univ. -Prof. Dipl. Ing. Dr. techn. Wolfgang Slany

Supervisor: Dipl. Ing. Dr. techn. Karl Voit

Graz, February 2013

This document was compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Contents

Abstract

Since usability evaluation has become an important element of software development, it is crucial to manage and process all generated usability testing data efficiently. This thesis compares two approaches in the field of evaluating HCI user experiments. It is shown, which input data, generated during these user experiments, is needed and which results can be generated in order to automatically evaluate qualitative and quantitative aspects.

In this work the functionality of the developed tool-kit is shown with the aid of the conducted formal experiments for the software *tagstore*. Thereby the process of automatic evaluation of a user logfile is described in detail. It is pointed out, that a general user logfile and a finite state machine for every use case are needed to be able to evaluate this logfile automatically and time analysis and other desired results can be derived.

This thesis also describes considerations regarding the re-usability of this tool-kit and automation of the evaluation process.

1 Introduction

Usability evaluation is often conducted with the help of user experiments. With a higher number of test persons, lots of data is generated that has to be processed. To keep the effort for the evaluation as low as possible, tools and methods are required, which can deal with data verification, processing and evaluation automatically.

This thesis describes, how the automatic evaluation of user logfiles can be realized with the help of intermediate language and compiler.

In Chapter 2 two similar approaches are described in detail. It is shown, which input data is needed and which results can be produced.

In Chapter 3 the process of the compiler development is discussed. This chapter points out

- How the intermediate language should be defined and how the corresponding finite state machines look like
- How the data processing is done
- How and which results can be calculated
- What needs to be considered regarding the reusability of the compiler
- What needs to be considered regarding the automation of the evaluation process.

2 Comparison with Corresponding Approaches

2.1 Automatic Mental Model Evaluator

AMME – Automatic Mental Model Evaluator (Rauterberg, 1992) – is a program for automatic analysis of Human-Computer interaction logfiles. It was developed to analyze observed sequences of decisions and actions of a user during completion of a task. Information that can be extracted from the user's decisions and actions are:

- Cognitive model of a user
- Individual problem solution strategies for a given task
- Underlying decision structure.

AMME is able to process this data automatically and derive

- An extracted net description of the task dependent decision model
- A complete state transition model
- Different quantitative measures of the decision behaviour.

2.1.1 Description of the Analyzing Tool-Kit

The analyzing tool-kit can be separated in three main parts (see Figure 2.1):

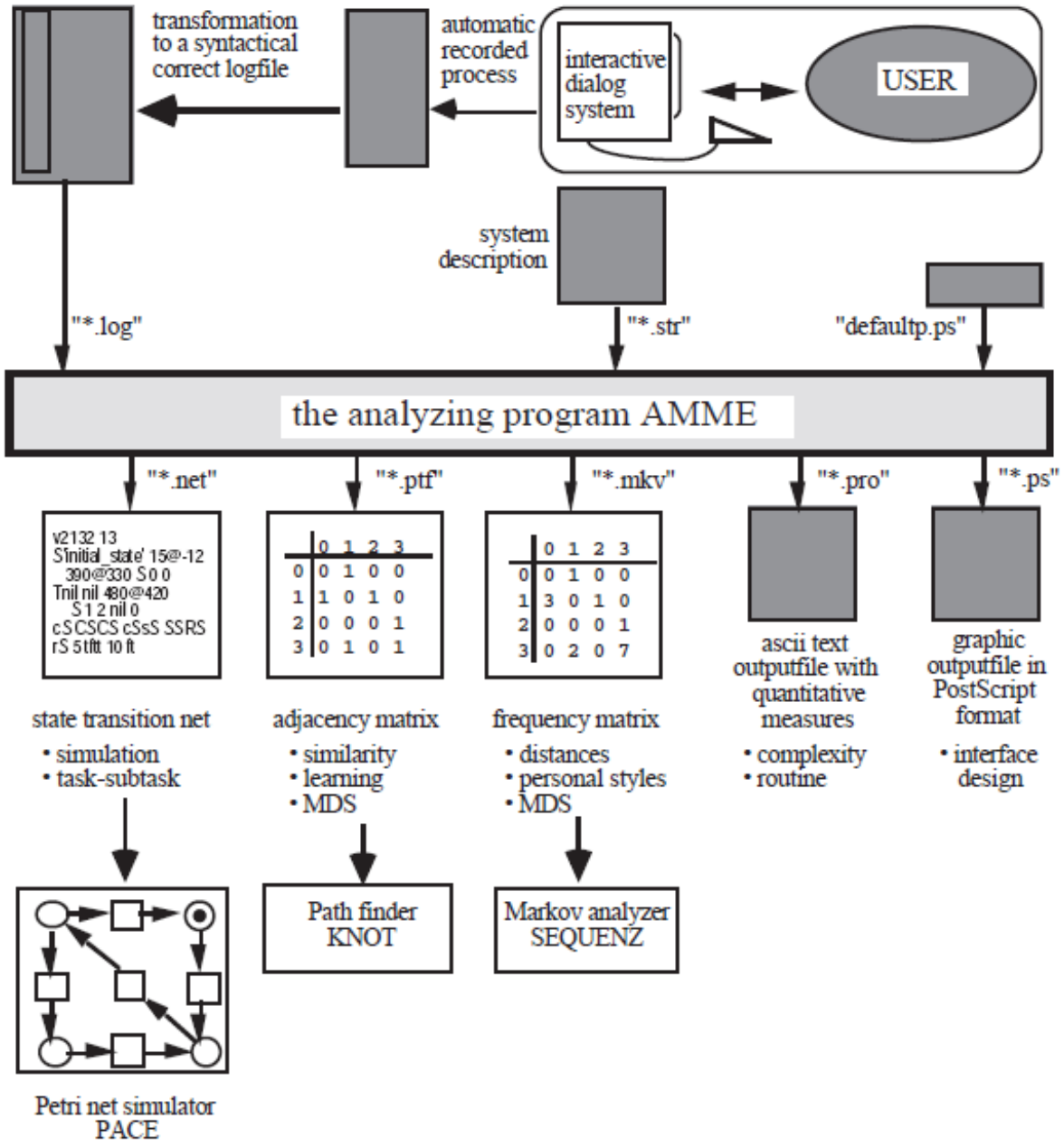

Figure 2.1: AMME tool-kit overview. [Source: Rauterberg (1992)]

The interactive dialog system and input files for AMME With the interactive dialog system and a logging feature, logfiles are generated automatically. These logfiles contain a process description of the user's task solving strategy. The generated logfile needs to be transferred to a syntactically correct logfile. This can be done either automatically or manually. It is also possible to create this logfile completely manual by an investigator. A second file contains the description of the system. The investigator has to define a state list and a pre-post state matrix. The state list needs to contain all relevant states of the investigated system. Which state is relevant and which not, is decided by the investigator. The pre-post state matrix contains a description of all allowed user decisions and their corresponding actions, which change the actual system state.

The analyzing program AMME AMME generates the net of the task processes from the input files and calculates quantitative measures. First the frequency of all observed elementary processes is counted to analyze first order Markov chains[1]. The resulting frequency matrix is written in one of the five produced output files. AMME is now able to combine all observed elementary processes and to build a Petri net[2]. This process is called folding. An example of folding a sequence can be found in Figure 2.2.

Result files which are produced by AMME and used for further analysis AMME creates five output files (see Figure 2.1).

- "*.net" – state transition net

  This file contains the resulting Petri net. It can be further analyzed with PACE[3], which is a Petri net simulator, consisting of a graphic editor and an interactive simulator with graphic animation.

- "*.ptf" – adjacency matrix

  This file contains an adjacency matrix, which can be analyzed by the net analyzing program KNOT[4]. With the help of KNOT it is possible to compute the similarity between pairs of nets.

- "*.mkv" – frequency matrix

[1] http://en.wikipedia.org/wiki/Markov_chain – retrieved on 2012-02-20
[2] http://www.petrinets.info – retrieved on 2012-02-20
[3] http://www.ibepace.com – retrieved on 2012-02-20
[4] http://interlinkinc.net/KNOT.html – retrieved on 2012-02-20

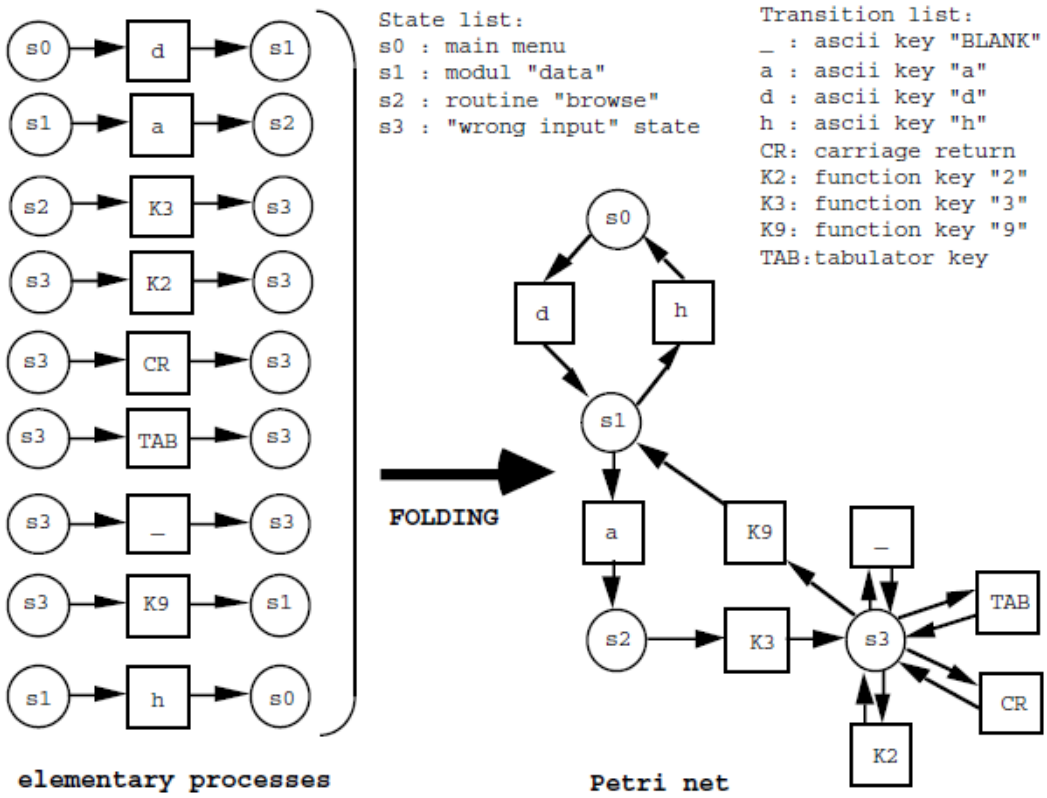

Figure 2.2: Sample processes and transformation into Petri net. [Source: Rauterberg (1992)]

logfile events	*content of the system description file*	*entries in *.log and*	*in *.str*
LOG_KEYBD : α	[α] ;# with α ∈ ASCII[032…127]	LOG_KEYBD : x	[x]
LOG_KEYBD :	[BL] ;# Blank,space	LOG_KEYBD :	[BL]
LOG_KEYBD : α	[ALL] ;# any α	LOG_KEYBD : r	[ALL]
LOG_KEYBD : <F β>	[F_β] ;# function key no.	LOG_KEYBD : <F_10>	[F_10]
LOG_KEYBD : <S 13>	[S_13] ;# ENTER, Return	LOG_KEYBD : <S_13>	[S_13]
LOG_MENUE : β	[M_β] ;# menue-no., β ∈ {0…99}	LOG_MENUE : 3	[M_3]
LOG_MESSAGE: β	[G_β] ;# message-no.	LOG_MESSAGE: 18	[G_18]
LOG_ERROR : β	[E_β] ;# error-no.	LOG_ERROR : 99	[E_99]
LOG_HELP : β	[H_β] ;# help-no.	LOG_HELP : 21	[H_21]
LOG_USRTIME: γ	# with γ ∈ {0,0…999,0} time unit	LOG_USRTIME: 11,0	nothing
LOG_SYSTIME: γ	# with γ ∈ {0,0…999,0} time unit	LOG_SYSTIME: 126,0	nothing

Figure 2.3: Overview of all possible logfile states. [Source: Rauterberg (1992)]

This file contains a frequency matrix, which can be analyzed by SEQUENZ[5]. This software compares sequences, which are produced by computer users.

- "*.pro" – ascii text outputfile with quantitative measures
- "*.ps" – graphic outputfile in PostScript format.

2.1.2 Structure of the Logfiles

User Logfile

In this logfile the task solving processes are recorded. In order to process the data correct, this file must have a syntactically correct form. Seven different states can be used:

LOG_KEYBD	Keybord entry, blank, space, function key number or return
LOG_MENUE	Menue number from 0 to 99
LOG_MESSAGE	Message number
LOG_ERROR	Error number
LOG_HELP	Help number
LOG_USRTIME	Time unit for user time
LOG_SYSTIME	Time unit for system time

Figure 2.3 shows all possible states, their various manifestations and corresponding representations in the system description file. The states LOG_USRTIME and

[5]U. Schmid and B. Meseke (1991)

output and input on the screen during each step	content of the logfile
initial state	LOG_KEYBD : <F 10>
UNIX(r) System V Release 4.0 (marshall)	LOG_MESSAGE: 1
login: rauterberg	LOG_KEYBD : r...g
Password:	LOG_KEYBD : *...*
Login incorrect	LOG_MESSAGE: 2
login: rauter	LOG_KEYBD : r...r
Password:	LOG_KEYBD : *...*
Last login: Wed Feb 7 19:16:57 from rota.ethz.ch	LOG_MENUE : 3
Sun Microsystems Inc. SunOS 5.4 Generic July 1994	LOG_MESSAGE: 4
Wed Feb 7 19:17:54 MET 1996	LOG_MESSAGE: 5
You have no mail.	LOG_MESSAGE: 24
marshall:/export/home/rauterberg!51> ls -l	LOG_KEYBD : l...l
total 2	LOG_MENUE : 16
drwxr-xr-x 7 rauter ifap 512 Feb 1 20:28 mac/	LOG_MENUE : 17
marshall:/export/home/rauterberg!52> x	LOG_KEYBD : x
initial state	

Figure 2.4: Example of user logfile for login and logout. [Source: Rauterberg (1992)]

LOG_SYSTIME are not considered in the system description file, but summed up during the parsing process.

As an example Figure 2.4 shows a concrete login and logout procedure on the screen of a Unix server and the corresponding content of the user logfile. Figure 2.5 shows the complete content of the user logfile of the above mentioned example.

If the analysis with AMME is done with a set of different task solving process descriptions, the net creation becomes an iterative procedure. That means, that after the analysis of all process descriptions is done once, it has to be done once more. This is necessary to be sure, that the calculation of all quantitative measures is based on the same system description.

System Description Logfile

The system description file is written by an investigator. First, a complete description of all relevant states needs to be entered. The relevance of a state is defined by the system behaviour or the investigators interpretation. Further the investigator needs to determine the granularity level. The lowest possible level is for example a mouse click. The right granularity level depends on the system behaviour. Furthermore the investigator needs to define a pre-post state matrix. That means, that all user decisions and their corresponding actions, which affect the system state (change from pre- to poststate), need to be defined. A state change can not only be caused by an user event, but also by a system reaction (system output, e.g. messages).

```
LOG_KEYBD  : <F 10>
LOG_USRTIME: 1,0
LOG_MESSAGE: 1
LOG_KEYBD  : r
LOG_USRTIME: 1,0
LOG_KEYBD  : a
LOG_USRTIME: 1,0
LOG_KEYBD  : u
LOG_USRTIME: 1,0
LOG_KEYBD  : t
LOG_USRTIME: 1,0
LOG_KEYBD  : e
LOG_USRTIME: 1,0
LOG_KEYBD  : r
LOG_USRTIME: 1,0
LOG_KEYBD  : b
LOG_USRTIME: 1,0
LOG_KEYBD  : e
LOG_USRTIME: 1,0
LOG_KEYBD  : r
LOG_USRTIME: 1,0
LOG_KEYBD  : g
LOG_USRTIME: 1,0
LOG_KEYBD  : <S_13>
LOG_USRTIME: 1,0
LOG_SYSTIME: 9,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
```

```
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : <S_13>
LOG_USRTIME: 1,0
LOG_SYSTIME: 49,0
LOG_MESSAGE: 2
LOG_SYSTIME: 8,0
LOG_KEYBD  : r
LOG_USRTIME: 1,0
LOG_KEYBD  : a
LOG_USRTIME: 1,0
LOG_KEYBD  : u
LOG_USRTIME: 1,0
LOG_KEYBD  : t
LOG_USRTIME: 1,0
LOG_KEYBD  : e
LOG_USRTIME: 1,0
LOG_KEYBD  : r
LOG_USRTIME: 1,0
LOG_KEYBD  : <S_13>
LOG_USRTIME: 1,0
LOG_SYSTIME: 9,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : *
LOG_USRTIME: 1,0
```

```
LOG_KEYBD  : *
LOG_USRTIME: 1,0
LOG_KEYBD  : <S_13>
LOG_USRTIME: 1,0
LOG_SYSTIME: 24,0
LOG_MENUE  : 3
LOG_SYSTIME: 2,0
LOG_MESSAGE: 4
LOG_SYSTIME: 7,0
LOG_MESSAGE: 5
LOG_MESSAGE: 24
LOG_SYSTIME: 41,0
LOG_KEYBD  : l
LOG_USRTIME: 1,0
LOG_KEYBD  : s
LOG_USRTIME: 1,0
LOG_KEYBD  :
LOG_USRTIME: 1,0
LOG_KEYBD  : -
LOG_USRTIME: 1,0
LOG_KEYBD  : l
LOG_USRTIME: 1,0
LOG_KEYBD  : <S_13>
LOG_USRTIME: 1,0
LOG_SYSTIME: 17,0
LOG_MENUE  : 16
LOG_MENUE  : 17
LOG_SYSTIME: 7,0
LOG_KEYBD  : x
LOG_USRTIME: 1,0
LOG_KEYBD  : <S_13>
LOG_USRTIME: 1,0
```

Figure 2.5: Complete content of user logfile for example login and logout. [Source: Rauterberg (1992)]

```
SM =              states {comments} transactions {comments} "END".
states =          "STATES" "=" statename {"," statename} ";".
transactions =    "TRANSITIONS" "=" {prestatename "=>" poststatename ";"}
                  "|".
prestatename =    statename.
poststatename =   statename "[" transitionname {"," transitionname} "]"
                  ";".
statename =       char {char | digit | "."}.
transitionname =  char | digit | "BL" | symbolchar "_" {number} | "ALL".
symbolchar =      "F" | "S" | "M" | "G" | "E" | "H".
comments =        "#" {" " | char | number} "CR".
number =          digit | digit digit.
char =            "a".."Z".
digit =           "0".."9".
```

Figure 2.6: Syntax definition for system description file. [Source: Rauterberg (1992)]

```
# list of all relevant dialog states of the interactive system

STATES  =

# states for correct input behavior
# [Note: the first state in this list defines the starting state for the analysis. One
#        attractive consequence is that any part of a logfile can be analysed through
#        updating this state descriptor list]
     initial.state ,
     login.state,
     systemprompt,
     listing.state,
     logout.state,

# states for incorrect input behavior

     login.wrong_input,
     ;

# list of all known transisitions
TRANSITIONS =
    initial.state     =>  login.state       [F_10] ;
    login.state       =>  systemprompt      [M_3] ;
    login.state       =>  login.wrong_input [G_2] ;
    login.state       =>  login.state       [ALL] ;
    listing.state     =>  systemprompt      [M_3] ;
    login.wrong_input =>  systemprompt      [M_3] ;
    systemprompt      =>  logout.state      [x] ;
    systemprompt      =>  listing.state     [M_16, M_17] ;
    systemprompt      =>  systemprompt      [G_4, G_5, G_24, ALL] ;
    logout.state      =>  initial.state     [S_13] ;
    |
END
```

Figure 2.7: System description file for login and logout example. [Source: Rauterberg (1992)]

Figure 2.6 shows the syntax definition for the system description file. Figure 2.7 shows the system description file for the above shown login and logout example.

2.1.3 Description of Results

With the files generated by AMME two different aspects can be distinguished:

Quantitative Aspects

Complexity – The complexity of a net can be calculated on the basis of the number of different states and the number of different transitions.

Routinization – This metric represents the ratio of the total number of transitions in the process description to the number of all necessary transitions.

Personality styles – If timestamps for a user are provided, the total task solving time and the number of different states per Petri net can be calculated. By dividing the total task solving time by the number of different states the mean duration time can be calculated. This value can be interpreted as the user's thinking time.

Qualitative Aspects

Diagnostic view – With the help of the graphical net representation special types of patterns could be recognised easily.

Task-subtask relationships – Possibility to check the own performance against the outcome obtained.

Learning processes – This measure allows the comparison between a net at a given learning stage and a reference net. As a consequence the status of the learning progress is updated easily.

2.2 Diagnostic Recorder for Usability Measurement (DRUM)

2.2.1 Description of Analyzing Software Tool

DRUM is a software tool, which was developed by Macleod and Rengger (1993). With the help of this tool, videos of usability tests can be analyzed efficiently. Measures and diagnostic data can be extracted as well. DRUM supports real-time and retrospective logging. Therefore an evaluator needs to identify and define events, which are of interest for the evaluation. These events, logged with a timestamp, are

then available in a database. All logged events can be edited and viewed at any later time. DRUM can be separated in four modules, each supporting different aspects of usability evaluation.

DRUM Evaluation Manager

The evaluation manager is the central part, where all data, collected at all stages of the evaluation process, comes together. The following data is available in the DRUM Evaluation Manager:

- Test persons
- Tasks – users tasks and predefined observable events
- Recording plans – technical arrangements for capturing raw data
- Evaluations – title and description of evaluations
- Video recording of evaluation sessions
- User logs created by DRUM
- Measures and metrics – calculated by DRUM analysing the logged task performance.

To ensure flexible compatibility, DRUM uses ASCII files for data storage.

DRUM Scheme Manager

DRUM provides a standard set of events, which may be of interest for the evaluator. The possibility of editing this set of events is also given. The DRUM Scheme Manager allows to:

- Edit, delete or rename existing events
- Add new events
- Group events
- Create hierarchical descriptions of the tasks.

Created schemes can be stored in the DRUM database for future use. Merging with other schemes is possible as well.

DRUM Recording Logger

With the help of this module the video related logs can be created and edited. Logging is possible in real-time and retrospective. The DRUM Recording Logger gives control of the video recorder and allows the evaluator to jump to already logged events. Recorded logs can be stored and retrieved from the DRUM database.

DRUM Log Processor

The DRUM Log Processor receives all log data from the database and is responsible for the calculation of performance measures and performance-based usability metrics.

2.2.2 Description of Results

The results calculated by the DRUM Log Processor are displayed in graphical and numerical form. For the reason of grouping data from different test persons and further statistical analysis, all calculated results can be saved in the database as well. Calculated results are:

- Task time – total time for each task
- Snag, help and search times – time a test person spends for having problems, seeking help or using the system unproductively
- Effectiveness – measure of how well the test person fulfilled the tasks
- Efficiency – this measure combines task time and effectiveness and can be understood as the rate of producing the task output
- Relative efficiency – this measure compares the efficiency of a test user with an expert's efficiency
- Productive period – the percentage of task time not spent in snag, help and search.

3 Compiler Development

In the previous chapter two examples of logfile analyzers were described. Unfortunately none of these tools could have been used for the analysis of the logged data from the tagstore[1] formal experiment[2]. Due to the low granularity level of logging events and the numerous thereof evolving metrics, the synthax for the logfiles and a finite state machine were developed.

3.1 Requirements

To be able to process the logged data in a structured and automatic way, the input files should be written as general as possible. In order to be able to create as universal logfiles as possible some guidelines should be followed:

- Logfiles should be stored as plain text files
- Only one event per line
- Each defined event starts with a timestamp.

For the evaluation of the logged user data from the formal experiment, which consisted of three parts (filing and re-finding with tagstore condition and hierarchy condition), a finite state machine and a specific language were defined for every part.

Filing with tagstore

Figure 3.1 shows the finite state machine for the filing task with tagstore. There are four different states a user can reach while filing with tagstore:

[1] http://tagstore.org – retrieved on 2012-02-20

[2] https://github.com/novoid/2011-04-tagstore-formal-experiment – retrieved on 2012-02-20

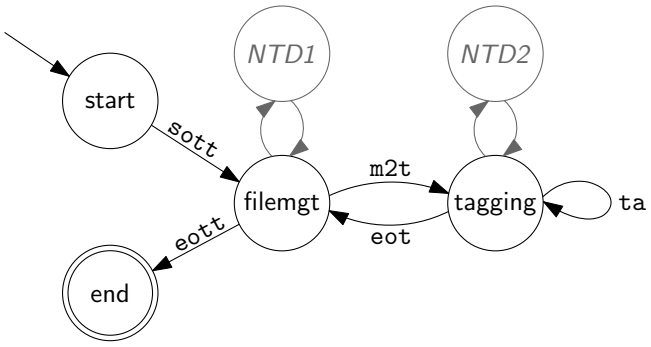

Figure 3.1: FSM for filing with tagstore. [Source: Voit (2012)]

- `filemgt` (file management)
  This state is used when a user copies or moves files from the initial directory to the tagstore storage.
- `tagging`:
  The state `tagging` is used when a user is in the stage of tagging files, which are in the tagstore storage. This state is left, when all files in the storage are tagged.
- `NTD1` (Non Task Distractions 1):
  This state can be reached from `filemgt` (file management) and means that a user is distracted from things which are not part of the test like viewing the content of the files again.
- `NTD2` (Non Task Distractions 2):
  This state can be reached from `tagging` and means that a user is distracted from things, which are not part of the test.

The events for this task are defined as follows:

<mm:ss> sott	Start of tagstore task
<mm:ss> m2t <nr files>	Copy or move <nr files> files to tagstore storage
<mm:ss> ta <nr tags>	Assign <nr tags> to a file
<mm:ss> eot	End of tagging (All files in the tagstore storage were tagged)
<mm:ss> eott	End of tagstore task.

Here is a simple example of a tagstore task log file:

00:42 sott
01:17 m2t 3
01:22 ta 2
01:26 ta 3

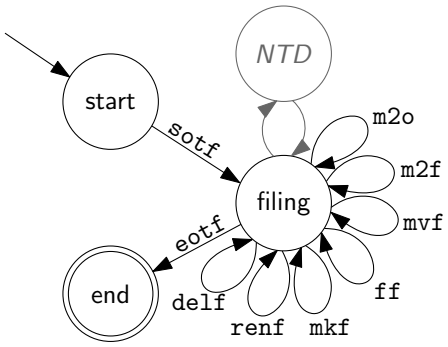

Figure 3.2: FSM for filing with hierarchy. [Source: Voit (2012)]

01:37 ta 1
01:38 eot
02:02 m2t 1
02:06 ta 1
02:07 eot
02:08 eott

Filing with Hierarchy

Figure 3.2 shows the finite state machine for the filing task with hierarchy. There are two different states a user can reach while filing with hierarchy:

- `filing`:
  This state is used when a user is filing items using hierarchy.

- `NTD` (Non Task Distractions):
  This state can be reached from `filing` and means that a user is distracted from things which are not part of the test.

The events for this task are defined as follows:

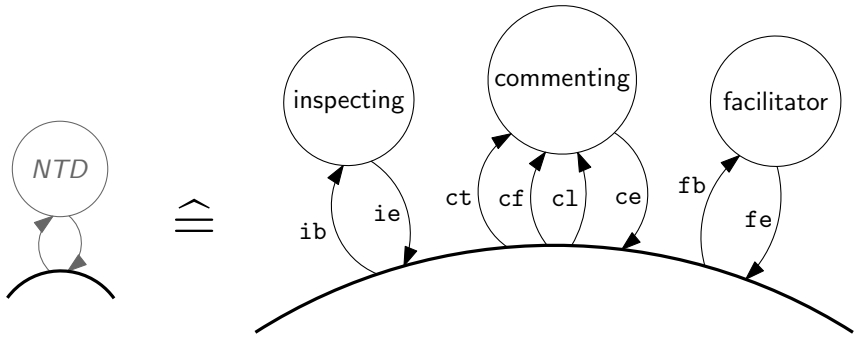

Figure 3.3: FSM for non task distractions. [Source: Voit (2012)]

<mm:ss> sotf	Start of folder task
<mm:ss> m2o <nr files>	Copy or move <nr files> from temporary directory to the target directory
<mm:ss> m2f <nr files> <folder>	Copy or move <nr files> from temporary directory to a sub folder in the target directory
<mm:ss> mkf <folder>	Create a new folder
<mm:ss> renf <folder1> <folder2>	Rename <folder1> to <folder2>
<mm:ss> delf <folder>	Delete <folder>
<mm:ss> ff <nr files> <folder>	Move <nr files> into <folder>
<mm:ss> mvf <nr folders> <folder>	Move <nr folders> folder(s) into <folder>
<mm:ss> eotf	End of folder task.

Here is a simple example of a folder task log file:

34:02 sotf
34:33 mkf Music
34:37 mkf Videos
34:48 mkf Images
36:03 m2f 3 Videos
36:26 renf Images "My pictures"
36:33 m2o 2
37:15 ff 2 "My pictures"
37:58 mvf 2 "My pictures"
38:02 delf Music
38:05 eotf

Non Task Distractions:

Figure 3.3 shows the finite state machine for all possible non task distractions. There are three types of events, which can distract the user from going on with the task:

- `inspecting`:
  The user can not remember what an item is about and therefore views the content of this file again.
- `commenting`:
  This state is reached, if someone gives a comment on anything and distracts the test person. Comments can possibly be made by the test person, facilitator or logger.
- `facilitator`:
  This state is entered, if the facilitator takes over control during a task (e.g. for solving a problem).

The events for the distractions are defined as follows:

<mm:ss> ib	Beginning of file inspection(s)
<mm:ss> ie	End of file inspection(s)
<mm:ss> c(f|t|l) <comment(s)>	Comment(s) of facilitator (f), test person (t) or logger (l)
<mm:ss> ce	End of comment(s)
<mm:ss> fb <comment(s)>	Facilitator takes over
<mm:ss> fe	Facilitator is finished and user starts again.

Distractions can occur during every other event.
Please note, that the times from state `NTD` were deducted from the times of the corresponding states in which these distractions occurred. The net times of each state, which are used for calculating the results, are computed as follows:

- Filing with tagstore:
  Net `filemgt` (file management) = Gross `filemgt` (file management) - `NTD1`
  Net `tagging` = Gross `tagging` - `NTD2`
- Filing with hierarchy:
  Net `filing` = Gross `filing` - `NTD`

Re-finding Task

Figure 3.4 shows the finite state machine for the re-finding task. The state `refind` is entered immediately from the beginning, until all tasks are done.

The events for the re-finding task are defined as follows:

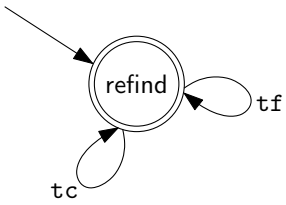

Figure 3.4: FSM for filing with hierarchy. [Source: Voit (2012)]

<mm:ss> tf <task nr> (<minutes>:)<ss.ds> <nr mouse clicks> (<nr keys>)
User finished task <task nr> in (<minutes>:)<ss.ds>, needed <nr mouse clicks> mouse clicks and used (<nr keys>) keys

<mm:ss> tc <task nr> (<minutes>:)<ss> (<comment>)
User skipped or canceled the task after (<minutes>:)<ss>. An optional comment can be added.

Here is a simple example of a re-finding task log file:

01:05 tf 4 05.62 2 0
01:37 tf 5 20.43 2 0
01:53 tf 6 06.39 2 0
02:20 tf 7 16.25 3 0
04:31 tc 8 45.08 "File not found"
04:49 tf 9 05.10 2 0

3.2 Parser Design

In order to analyze the log files automatically a compiler for every type of log file was developed. With the help of the compilers two goals should be achieved:

- Check, if the log files are lexically and syntactically correct
- Measure how much time a user spent in which state and extract metrics of interest.

As long as errors in the log occur, no results are created. Hints for debugging the logfiles are given by the analyzer as well. Once the log files are errorless, results are being computed and written to a csv file.

LogAnalyzerTagstore

This compiler analyzes the log files of the tagstore filing task which are the input for this analyzer. The compiler generates a csv file which contains the following results:

- Overall time
  Gross time for overall tagstore filing task (incl. `NTD`).
- Overall time without distractions
  Net time for overall tagstore filing task (excl. `NTD`).
- Tagging time
  Gross time user spent in state `tagging` (incl. `NTD2`).
- Tagging time without distractions
  Net time user spent in state `tagging` (excl. `NTD2`).
- FMGT time
  Gross time user spent in state `filemgt` (file management) (incl. `NTD1`).
- FMGT time without distractions
  Net time user spent in state `filemgt` (file management) (excl. `NTD1`).
- Sum of distractions
  Sum of all NTD.
- Tagging distractions
  Sum of all distractions which occurred during the state `tagging` (`NTD2`).
- FMGT distractions
  Sum of all distractions, which occurred during the state `filemgt` (file management) (`NTD1`).
- Commentary time
  Sum of all distractions which result from comments of test person, facilitator or logger.
- TP Inspection time
  Sum of all distractions which result from viewing test items during the filing task.
- Facilitator time
  Sum of all distractions which occurred when the facilitator took over.
- Sum of tags
  Sum of used tags for all test items.
- Avg Tags/Item
  Average of used tags per test item.

- List # tags/item
  List with the number of tags which were used for item 1 .. n.
- List of time user spent for tagging/item
  List of time (in s) user spent for tagging item 1 .. n.
- # m2t
  Sum of occurrences of the event: m2t.

LogAnalyzerFolder

This compiler analyzes the log files of the hierarchy filing task which are the input for this analyzer. The compiler generates a csv file which contains the following results:

- Overall time
  Gross time for overall hierarchy filing task (incl. `NTD`).
- Overall time without distractions
  Net time for overall hierarchy filing task (excl. `NTD`).
- FMGT time
  Gross time user spent in state `filemgt` (file management) (incl. `NTD`).
- FMGT time without distractions
  Net time user spent in state `filemgt` (file management) (excl. `NTD`).
- Sum of distractions
  Sum of all `NTD`.
- FMGT distractions
  Sum of all distractions which occurred during the state `filemgt` (file management) (`NTD`)
- Commentary time
  Sum of all distractions which result from comments of test person, facilitator or logger.
- TP Inspection time
  Sum of all distractions which result from viewing test items during the filing task.
- Facilitator time
  Sum of all distractions which occurred when the facilitator took over.
- Number of created folder
  Sum of created folders.

- # M2F
  Sum of occurrences of the event: m2f.
- # MVF
  Sum of occurrences of the event: mvf.
- # RENF
  Sum of occurrences of the event: renf.
- # DELF
  Sum of occurrences of the event: delf.

LogAnalyzerRefinding

This compiler analyzes the log files of the re-finding task, which are the input for this analyzer. The compiler generates a csv file, which contains the following results:

- Mouse clicks
  List with the number of mouse clicks a test user needed to finish task 1 .. n.
- Time per Task
  List with the times (in (mm:)ss.ds) a test user needed to finish task 1 .. n.

3.3 Re-Usability

Regarding the re-usability of a logfile compiler some guidelines should be concerned:

- No formal restrictions for the logfiles
- Easy adoption/extension of event definitions
- Easy adoption/extension of presentation of results
- Easy adoption/extension of calculation of measures.

In order to be able to reuse the results whenever needed, these results should be exported in an very clear and self describing way. If possible a csv file with a header should be used to store all relevant data. The advantage of this file format is, that nearly all tools can deal with this file type and therefore further processing is easy.

3.4 Automation

For a big amount of test persons it is important to optimize the whole evaluation process regarding the automation. Following steps would be necessary to get a completely automated tool-kit:

- Automatic logging of pre-defined events during the evaluation
- Import and evaluation of all logfiles at once
- Export of all results at once.

List of Figures

Bibliography

Macleod, Miles and Ralph Rengger (1993). "The development of DRUM: A software tool for video-assisted usability evaluation." In: *In Proceedings of HCI'93*. Cambridge University Press, pp. 293–309.

Rauterberg, M. (1992). *AMME: an "Automatic Mental Model Evaluator" to analyse user behaviour recorded on logfiles*. Berlin, Germany.

Voit, Karl (Nov. 2012). "TagTrees: Improving Personal Information Management using Associative Navigation." PhD thesis. Graz, Austria: Graz University of Technology.