LOGDIG Log File Analyzer for Mining Expected Behavior from Log Files

Esa Heikkinen, Timo D. Hämäläinen

Tampere University of Technology, Department of Pervasive computing,
P.O. Box 553, 33101 Tampere, Finland

Email: esa.heikkinen@student.tut.fi
timo.d.hamalainen@tut.fi

Abstract. Log files are often the only way to identify and locate errors in a deployed system. This paper presents a new log file analyzing framework, LOGDIG, for checking expected system behavior from log files. LOGDIG is a generic framework, but it is motivated by logs that include temporal data (timestamps) and system-specific data (e.g. spatial data with coordinates of moving objects), which are present e.g. in Real Time Passenger Information Systems (RTPIS). The behavior mining in LOGDIG is state-machine-based, where a search algorithm in states tries to find desired events (by certain accuracy) from log files. That is different from related work, in which transitions are directly connected to lines of log files. LOGDIG reads any log files and uses metadata to interpret input data. The output is static behavioral knowledge and human friendly composite log for reporting results in legacy tools. Field data from a commercial RTPIS called ELMI is used as a proof-of-concept case study. LOGDIG can also be configured to analyze other systems log files by its flexible metadata formats and a new behavior mining language.

Keywords: Log file analysis, data mining, spatiotemporal data mining, behavior computing, intruder detection, test oracles, RTPIS, Python

1 Introduction

Log files are often the only way to identify and locate errors in deployed software [1], and especially in distributed embedded systems that cannot be simulated due to lack of source code access, virtual test environment or specifications. However, log analysis is no longer used only for error detection, but even the whole system management has become log-centric [2].

Our work originally started 15 years ago with a commercial Real Time Passenger Information System (RTPIS) product called ELMI. It included real-time bus tracking and bus stop monitors displaying time of arrival estimates. ELMI consisted of several mobile and embedded devices and a proprietary radio network. The system was too heterogeneous for traditional debugging, which led to log file analysis as the primary method. In addition, log file analysis helped discovering the behavior of some black-box parts of the system, which contributed to the development of open ELMI parts.

The first log tools were TCL scripts that were gradually improved in an ad-hoc manner. The tooling fulfilled the needs very well, but the maintenance got complicated

and the tools fit only for the specific system. This paper presents a new, general purpose log analyzer tool framework called LOGDIG based on the previous experience. It is purposed for logs that include temporal data (timestamps) and optionally system-specific data (i.e. spatial data with coordinates of moving objects).

Large embedded and distributed systems generate many types of log files from multiple points of the system. One problem is that log information might not be purposed for error detection but e.g. for business, user or application context. This requires capability to interpret log information. In addition, there can be complex behaviors like a chain of sequential interdependent events, for which simple statistical methods are not sufficient [1]. This requires state sequence processing. LOGDIG is mainly intended for *expected behavior* mining.

State-of-the art log analyzers connect the state transitions one by one to the log lines (i.e. events or records). LOGDIG has an opposite new idea, in which log events do not directly trigger transitions but events are searched for states by special state functions. The benefit is much more versatile searches and inclusion of already read old log lines in the searches, which is not the case in state-of-the-art. LOGDIG includes also our new Python based language Behavior Mining Language (BML), but this is out of the scope of this paper.

The new contributions in this paper are i) New method for searching log events for state transitions, ii) LOGDIG architecture and implementation iii) Proof-of-concept by real RTPIS field data.

This paper is organized as follows. Section 2 describes the related work and Section 3 the architecture of LOGDIG. Section 4 presents an RTPIS case study with real bus transportation field data. The paper is concluded in Section 5.

2 Related work

We focus on discovering the realized behavior from logs, which is required to figure out if the system works as expected. The behavior appears as an execution trace or sequential interdependent events in log files. The search means detecting events that contribute to the trace described as *expected behavior*.

A typical log analyzer tool includes metamodels for the log file information, specification of analytical tasks, and engines to execute the tasks and report the results [2]. Most log analyzers are based on state machines. The authors in [5] conclude that most appropriate and useful form for a formal log file analyzer is a set of parallel state machines making transitions based on lines from the log file. Thus, e.g. in [1], programs are validated by checking conformity of log files. The records (lines) in a log file are interpreted as transitions of the given state machine.

The most straightforward log analyzing methods is to use small utility commands like **grep** and **awk**, spreadsheet like Excel or database queries. More flexibility comes with scripts, e.g. Python (NumPy/SciPy), Tcl, Expect, Matlab, SciLab, R and Java (Hadoop). There are also commercial log analyzers like [3] and open source log analyzers like [4]. However, in this paper we focus on scientific proposals in the following.

Valdman [1] has presented a general log analyzer, and Viklund [5] analysis of debug logs. Authors in [6] have presented execution anomaly detection techniques in distributed systems based on log analysis. Execution anomalies include work flow errors and low performance problems. The technique has three phases: at first abstracting log files, then deriving FSA and next checking execution times. The result is a model of behavior. Later it can be compared whether the learned model was same or not than current model to detect anomalies. This technique does not need extra languages or data to configure analyzer. A database based analysis is presented in [7] for employing a database as the underlying reasoning engine to perform analyses in a rapid and lightweight manner. TBL-language is presented in [8] for validating expected behaviors in execution traces of system. TBL is based on parameterized patterns. Domain specific LOGSCOPE language is presented in [9] that is based on temporal logic.

LFA [10] has been developed for general test result checking with log file analysis, and extended by LFA2 [11; 12] with the idea of generating log file analyzers from C++ instead of Prolog. That improved general performance of LFA and allowed to extend the LFAL language by new features, such as support for regular expressions. Authors in [13] have presented a broad study on log file analysis. LFA is clearly closest to our work and most widely reported.

Table 1 presents comparison between our work and LFA/LFA2. The classification criteria is modified from [1] and for brevity we exclude detailed description of each. To conclude, the main difference to LFA/LFA2 is that it connects transitions directly to the events (lines) of the log files. LOGDIG has a completely different approach enabling more versatile searches, like the search from old (already read) lines or the search “backward”. New system-specific search features are also easier to add.

Table 1. Comparison of LOGDIG and LFA/LFA2 analyzer

Feature	LOGDIG	LFA/LFA2
1. Analyzed system and logs		
1. System type	Distributed	Distributed
2. Amount of logs	Not limited	1
3. size of logs	Not limited	Not limited
4. formats of logs	Not limited	Limited
2. Analyzing goals and results		
1. Processing	Batch job	Batch job
2. type of knowledge	Very complex	Complex
3. Results	SBK, RCL, test oracle	Test oracle
3. Analyzator		
1. technology or formalism of engine	State-machine	State-machine
2. using event indexing	Time, Id	No
3. Search aspects (SA)	Index, data, SSD	Data
4. Adjustment of accuracy of SA	Yes	Yes
5. pre-processing (and language)	Yes (PPL)	No
6. analyzing language	BML	LFAL
7. versatile reading from log	By index not limited	Only forward
8. operational modes	Multiple-pass	Single-pass
9. state-machine-based engine	Yes	Yes
1. state machines	1	Not limited
2. state-functions	ES algorithm	No
3. state transitions	Limited (F,N,E)	Not limited
4. state transition functions	Yes	Yes
5. variables	Yes	Yes
4. Analyzator's structure and interfaces		
1. implementation	Python (Tcl)	Prolog/C++
2. extensibility	Via SBK, BMS, SSC	?
3. integratability	Command line	?
4. modularity	BMU,ESU,SSC,SSD,lib	Lib
5. Analyzator's other features		
1. robustness (runtime)	Exit transition	Error transition
2. user interface	Command line	?
3. compiled or interpreted	Interpreted	Compliled
4. speed	Medium	Fast

3 LOGDIG architecture

Fig. 1 depicts the overall architecture and log processing of LOGDIG. The analysis process refines raw data from the log files (bottom) into information (middle) and finally to knowledge (top). The straightforward purpose is to transform the dynamic behavior data from the log files into **static behavior knowledge** (SBK file). This can be thought of linking many found events from many "rows" of log files into one "row" of the SBK file. The static knowledge can be reported and further processed by legacy visualization, database and spreadsheet tools like Excel. Besides SBK, LOGDIG produces a Refined Composite Log called RCL file. Its purpose is to link found data on the log files together into one composite, human readable text format described in BML language. This helps simple keyword-based searches for reporting and further analysis.

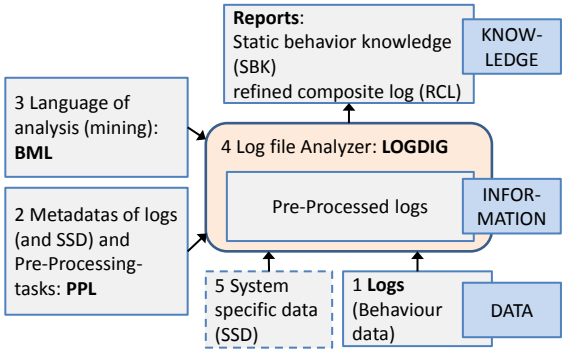

Fig. 1. Overview of LOGDIG analyzer log processing.

The LOGDIG processing has two main phases: log data pre-processing and behavior mining. Pre-processing transforms the original log files into uniform, mining-friendly format, which is defined by our own Pre-Processing Language (PPL). PPL is not discussed further in this paper, but we use the common **csv** format with named columns as output of the transform. The behavior mining phase uses our BML language to define searching the events of expected behavior.

Logs may consist of rows of unstructured text that can be parsed with regular expressions (Python). Timestamp is mandatory in every line. Pre-processed logs are structured in rows and columns as csv files. The first row is header line that includes column (variable) names. The columns are row number, timestamp and log file specific data.

Sometimes analyzing needs systems specific utility data to make decisions in behavior mining phase. This is called System Specific Data (SSD) and it is not log data. SSD files can come from system documentation and databases without any standard structure and format as in our case study. SSD can be e.g. spatial data like position boundary boxes for bus stops in RTPIS. SSD is parsed using System Specific Code (SSC) written in Python.

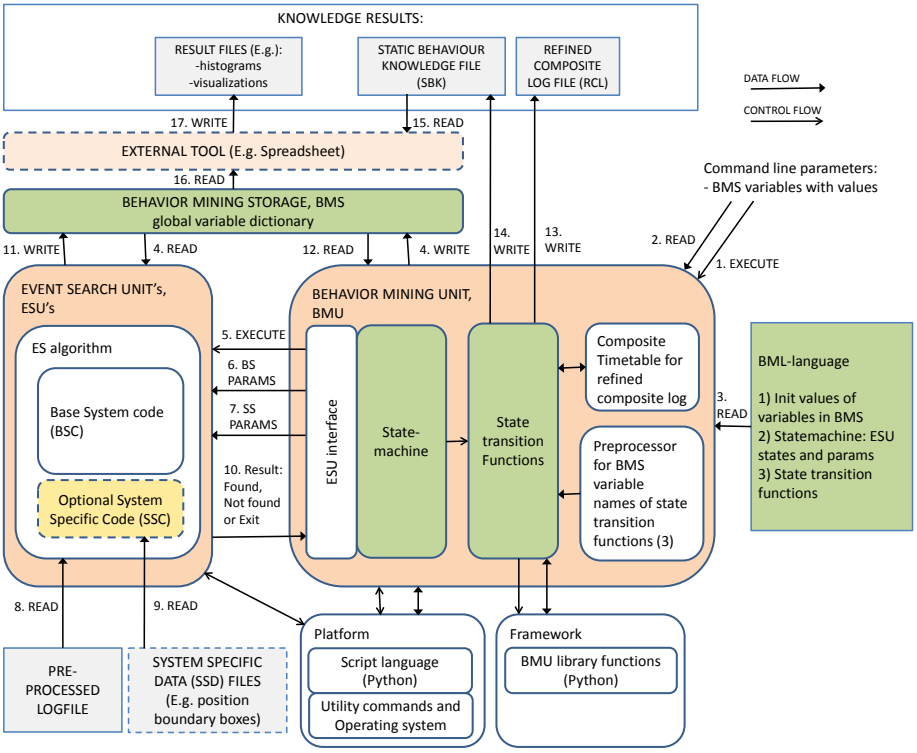

Fig. 2. Detailed architecture of behavior mining phase of LOGDIG.

The detailed behavioral mining process is presented in Fig. 2. We first describe the concepts and functional units. There are two main functional units: Behavior Mining Unit (**BMU**) and Event Search Unit (**ESU**) and also one global data structure named as Behavior Mining Storage (**BMS**). There can be many ESU's, one BMU and one BMS in LOGDIG.

BMS is a global and common dictionary type data structure, which includes variable names and their values used in LOGDIG. Initial BMS can be described in BML-language or given in command line parameters. It is possible to add new variables and values to BMS on the fly. Used variables can be grouped as follows: command line, internal, and result variables. The result variables are grouped as identify, log, time, searched, calculated, counter, error flags and warning flag variables (See example in table 2). BMS can be implemented as shared memory structure wherein it is very fast and it can be used in many (external) tools and commands even in real time.

BMU consists of five parts: pre-processor of variable names, state-machine, state-transition functions, ESU interface and composite timetable. **Pre-processor** converts variable names in state transition functions of a BML code to be used in Python. State machine is the main engine of BMU that is described in BML language. State transi-

tion functions are called from the state machine and they are also described in BML-language.

ESU interface connects the state machine to ESU. It converts execute command and parameters suitable to ESU and converts the result of ESU for the state machine. The ESU interface can be modified depending on how ESU has been implemented and where it is located, compared to BMU. ESU can be a separate command line command and it can be even located on separate computers. This gives interesting possibilities for the future to decentralization and performance.

A **composite timetable-structure** is for the Refined Composite Log (RCL) file to collect together all time indexed "print"-notifications from state transitions. The time-table is needed because all searched data from log files are not always read in time (forwarding) order, because BMU has capability to re-read old lines from same log and also to read older line from other logs.

BML language presents the expected behavior with certain accuracy and it is straightforward representation of state machine, (ESU) states with input parameters and state transition functions. It can also include initialization of BMS variables and values. State transition functions can include Python code and BMS variables. Because BML is out of the scope of this paper, we do not go into details.

ESU is a state of the state machine and it includes state function which is named Event Search (ES) algorithm. ES algorithm includes a Basic System Code (BSC) and optionally a System Specific Code (SSC). The BSC is the body of ES algorithm and SSC is an extension to BSC. BSC reads Basic System (BS) parameters from the state machine and on their basis start searching from log. If SSC is in use (set in mode parameter), also System Specific (SS) parameters are read.

Base System (BS) parameters are needed to set the search mode, log file name expression, searched BMS log variable names, time column name, as well as start and stop time expressions for ES algorithm. **System Specific (SS) parameters** are used in SSC-part of ES algorithm. E.g. in case of spatial data like in our case study, SS parameters are: latitude and longitude column name and also SSD filename expression.

Each search mode has the common task: searches an event between the start and the stop time (ESU time windows in figure 5) that values of variable names are same in log and BMS.

The search modes can be:

- "EVENT:First": Searches the first event from the log
- "EVENT:Last": Searches the last event from the log

In the case of SSC, search modes can also be System Specific (SS) modes (as in our case study):

- "POSITION:Entering": Searches object's entering position to boundary box area (desciped in SSD) from the log
- "POSITION:Leaving": Searches object's leaving from boundary box area (described in SSD) from the log

3.1 Mining process

The mining process is described in figure 2. The user of LOGDIG starts the BMU execution (1), which reads command line parameters (2) and BML language (3) as input. On their basis BMU writes (4) BMS variable values directly or via state transition function, and executes (5) ESU with search BS parameters (6). If SS mode is in use in current ESU, also SS parameters (7) are read. Then ESU reads (8) pre-processed log file. If SS mode in use, it reads also (9) SSD file. Then starts the searching.

After ESU has finished the searching, it returns (10) the result of the search, which can be "Found" (F), "Not found" (N) or "Exit" (E). If "Found", ESU writes (11) found variables from the log file to the BMS variables. Then, BMU reasons what to do next based on the action attached to the result. This is described in BML language. Depending on the action, BMU runs possible state transition function which can read (12) BMS variables. Based on the action, a transition function may write one row to (13) RCL or (14) SBK **knowledge-result** files.

After that BMU normally executes (5) a new search by starting ESU with new parameters. BMU exits the mining phase if there are no searches left or "Exit" has been returned. "Exit" is a problem exception that should never happen, but if it occurs, user is notified and LOGDIG is closed cleanly.

When BMU has completed, the SBK file can be read (15) by an **external tool** (like spreadsheet) to write (17) better visualizations from the knowledge results. All tools supporting the csv format can be used. Another option is to read (16) directly data from the BMS variables by a suitable tool to write (17) other (visualization) knowledge-results.

State transition functions can use **framework BMU library** functions, like setting and writing RCL and SBK files and also calculating time differences between timestamp. It is also possible to add new library-functions depending on the needs.

Because LOGDIG analyzer has been implemented in Python and it works on top of any operating system, all features of Python and the operating system can be used as a **platform** in BMU and ESU.

4 Case study: EPT-case in ELMI

The real time passenger information system ELMI [14], deployed in Espoo, Finland in 1998 - 2009, displayed the waiting time of buses at bus stop monitors. ELMI included 300 buses and 11 bus stop monitors [15], 3 radio base stations, central computer system (CCS) and communication server (COMSE). Buses sent login of line and location information to CCS via radio base stations every 30 seconds. Then CCS calculated the waiting time values and sent them to the bus stop monitors via COMSE. Because all messages of the system went through the COMSE, it was also used for logging and running the LOGDIG analyzer. There are 3 types of original logs: CCS, COMSE and BUS. Every bus has own BUS log identified by bus number. In preprocessing phase of LOGDIG, original logs are divided message-specified log files: CCS_RTAT, CCS_AD, BUS_LOGIN and BUS_LOCAT. That makes analyzing faster and simpler.

We have named the main requirement as EPT (Estimated Passing Time) to see when the bus pass the stop. The *expected behavior* can include one or more EPT's.

Other definitions include RTAT (Real bus Theoretical Arrival Time), including bus number, line and monitor, as well as AD (Advanced or Delayed) compared to RTAT.

The sequence of requirements for one EPT case is: 1) a bus driver sent a login for a line in a bus terminal, 2) the bus started to drive from the terminal, 3) when the bus is in the line and if necessary (started, delayed or advanced of schedule) CCS calculated estimated waiting times and sent to target monitors, 4) the bus arrived to a bus stop and its monitor, and 5) COMSE removed the waiting time from monitor.

There are two SSD's (System Specific Data) in this case: 1) customer's requirement document that requires the error of estimation should be below 120 seconds and 2) boundary box areas of bus stop's from database of ELMI-system.

Various EPT cases are processed by the LOGDIG state machine shown in Fig. 4. It can search and analyze all EPT's in a time window of Expected Behavior (EB) for specific lines and monitor as described in Fig. 3. The time window can be set from the command line or BML language.

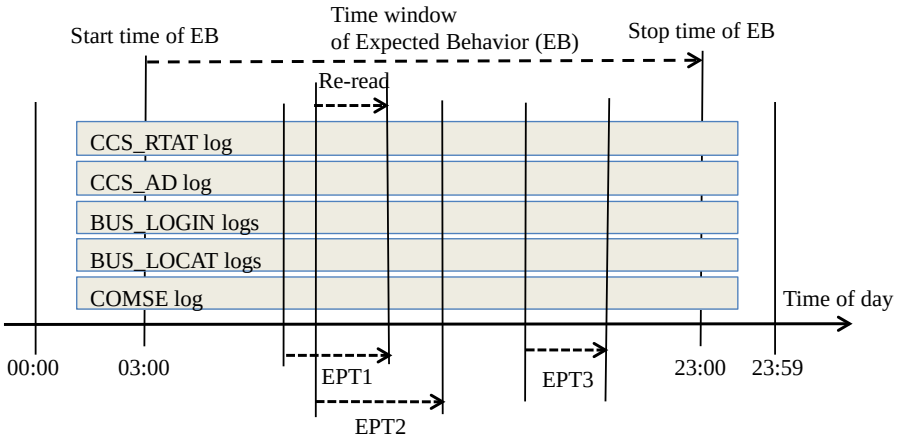

Fig. 3. Time window from 03:00 to 23:00 and 3 example EPT's in logs

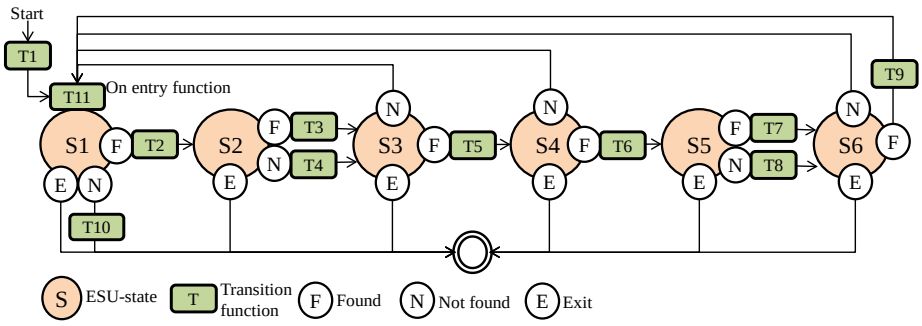

Fig. 4. State machine structure of the EPT's case study

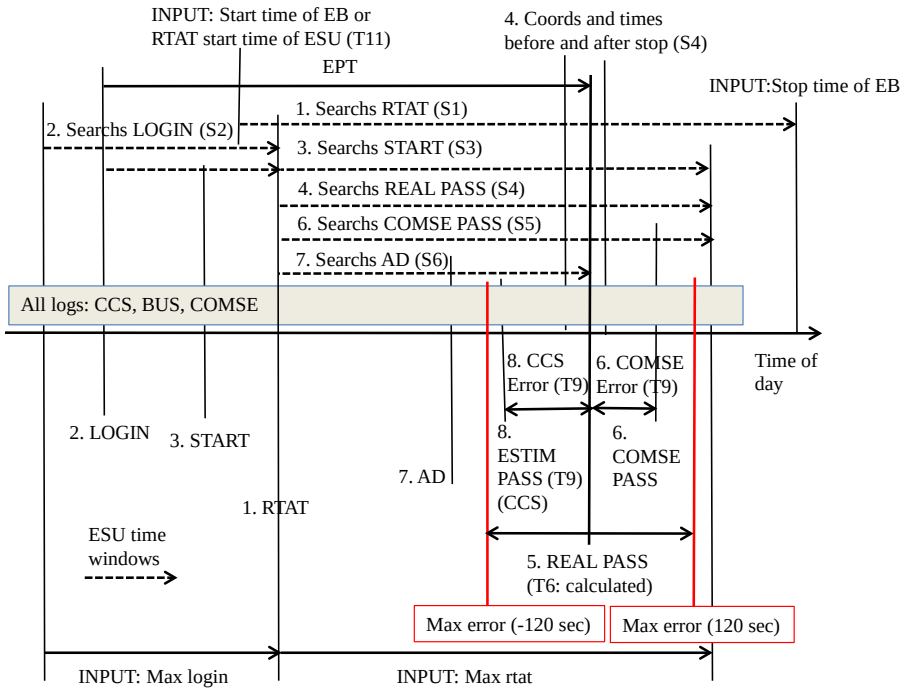

Fig. 5. Timing diagram for one EPT

Behavior and ESU-states of the state machine of LOGDIG in EPT's case is depicted in Fig. 5. Inputs of the EPT's case (from T1 transition function or command line) are start time, stop time, line, max login (means maximum time from LOGIN to RTAT) and max rtat (means maximum time from RTAT to PASS) and also maximum error values (SSD 1: -120 – 120 seconds) for the estimations of CCS and COMSE. The sequence is following:

- S1: Searches first RTAT message (inputs: line) from CCS_RTAT log file in given ESU time window (see Fig. 5):
  - On entry function (before searching), calls T11: Sets start time of ESU time window to search next RTAT message or original EB start time in first "round" of search
  - Found: Sets RTAT variables. Calls T2: Sets input variables for next state.
  - Not found: Calls T10: Prints final results.
- S2: Searches first LOGIN message (inputs: log-type, bus, line, direction) from BUS_LOGIN log file in given ESU time window:
  - Found: Sets LOGIN variables. Calls T3: Sets input variables for next state.
  - Not found: Calls T4: Sets input variables for next state.
- S3: Searches starting (leaving) place of the bus from terminal bus stop in given ESU time window from BUS_LOCAT log file (input: bus). This needs SSD 2 to check positions:

  - Found: Sets LOCAT variables. Calls T5: Sets input variables for next state.
  - Not found: -
- S4: Searches arriving place of the bus to the target bus stop in given ESU time window from BUS_LOCAT log file. This needs SSD 2 to check position:
  - Found: Sets LOCAT variables. Calls T6: Sets input-variables for next state. Calculates the real passing time of the bus
  - Not found: -
- S5: Searches first BQD message (inputs: bus, line) from COMSE log file in given ESU time window:
  - Found: Sets BQD variables. Calls T7: Sets input variables for next state.
  - Not found: Calls T8: Sets input variables for next state.
- S6: Searches last AD message (inputs: line, direction, bus) from CCS_AD log file in given ESU time window:
  - Found: Sets AD variables. Calls T9: Calculates errors of the estimated waiting time, writes one row to SBK file and writes RCL knowledge from the composite timetable structure to RCL file
  - Not found: -

There are exit transitions in every ESU state if a problem reading a log file occurs, e.g. the file is missing or there is a syntax error. Variables and their values (BMS variables in Fig. 2) comes directly from the log files and the command line initialization parameters, but there are also result specific variables in the SBK files. Result variables of SBK file are set in almost every state and state transition function and they are added (by one row) to SBK file at the end of successfully analyzing of one EPT in the last transition (T9 in Fig. 4). RCL knowledge is written in almost every state transition function to the composite timetable-structure (see Fig. 2) and in T9 they are added to RCL file. Searching time windows in every state can vary depending on (initializing variables and) variable values (found events timestamps) given from previous states.

Time indexing and possibility to read already processed log lines is utilized in two ways. See the re-read between EPT1 and EPT2 in Fig. 3. When searching different log files, RTAT-message from CCS_RTAT log is read before LOGIN message from BUS_LOGIN log even though LOGIN (EPT requirement step 1) comes before RTAT (EPT requirement step 2) in time. This way we can limit the complexity of the search and analyze only certain RTAT and its line or monitor. Within the same log file, RTAT or AD messages in many EPT cases exist. If we want to analyze only one monitor and its line, we will re-read the old log lines.

4.1 Case study results

The content of the SBK file in the ELMI case study is given in Table 2. There are described three example EPT's (bus drives in line 2132 and towards bus stop monitor 1031), which are also shown in Fig. 6. We consider a more detailed reading of the first EPT (EPT1). At 6:24:31 (LOGIN_MSG) in the terminal, the bus driver (number111) has logged in to line 2132 and its direction 1. Then at 6:25:01 (DRIVE) the

bus has started from the terminal. At 6:42:33 (RTAT_MSG) CCS has sent the first estimated arrival time (6:45:52, RTAT_VAL) to the monitor. At 6:45:03 (AD_MSG) CCS has sent the last time correction for the waiting time. AD_VAL -25 tells that the bus has been 25 seconds in advance of the schedule. At 6:45:40 (PASS) the bus has passed the bus stop that is the estimation of LOGDIG. COMSE has estimated the arrival time to be 6:46:02 (BQD_MSG). That means there has been 13s error (PASS_TIME_ERR) between the estimated arrival time (RTAT_AD_VAL: 6:45:27) and real arrival time (PASS: 6:45:40). The error to the COMSE estimation has been -22 seconds (BQD_TIME_ERR). Because absolute values of the errors are smaller than 120 seconds, error flag EPT_ERR is 0. Warning flags are 0, that means there have been found LOGIN and BQD messages from the logs. In this case the whole EPT has lasted from 6:24:31 (LOGIN_MSG) to 6:46:02 (BQD_MSG) in total of 21:31 minutes.

Table 2. Three example EPT's (in line 2132 and bus stop monitor 1031) of the SBK file

Knowledge variables			Knowledge data		
Group	Num	Variable	EPT1	EPT2	EPT3
Identify	1	SBK_ID	13	35	52
Log	2	BUS	111	451	228
	3	LINE	2132	2132	2132
	4	DIR	1	1	1
	5	SIGN	1031	1031	1031
Time	6	RTAT_SRCH_TIME	6:36:01	7:55:47	8:48:08
Searched	7	LOGIN_MSG (S2)	6:24:31	7:48:17	8:43:09
	8	DRIVE (S3)	6:25:01	7:48:47	8:44:10
	9	RTAT_MSG (S1)	6:42:33	7:55:47	8:48:08
	10	RTAT_VAL (S1)	6:45:52	7:59:02	8:51:18
	11	PASS (S4)	6:45:40	8:01:57	8:51:45
	12	BQD_MSG (S5)	6:46:02	8:02:17	8:52:08
	13	AD_MSG (S6)	6:45:03	8:01:48	8:51:09
	14	AD_VAL (S6)	-25	166	11
Calculated	15	RTAT_AD_VAL (T9	6:45:27	8:01:48	8:51:29
	16	PASS_LOGIN (T9)	1269	820	516
	17	PASS_DRIVE (T9)	1239	790	455
	18	PASS_RTAT (T9)	187	370	217
	19	PASS_TIME_ERR (T	13	9	16
	20	BQD_TIME_ERR (T	-22	-20	-23
Error flags	21	EPT_ERR	0	0	0
Warning	22	LOGIN_WARN (T4)	0	0	0
flags	23	BQD_WARN (T8)	0	0	0

The same EPT's has also been presented in a visual form in Fig. 6. The visual presentation has been generated from the SBK file using gnuplot and Tcl script language. The horizontal axis represents the time of day and the vertical axis time differences in relation to the passing time. The values can be directly found from the SBK variables numbers: 16-20.

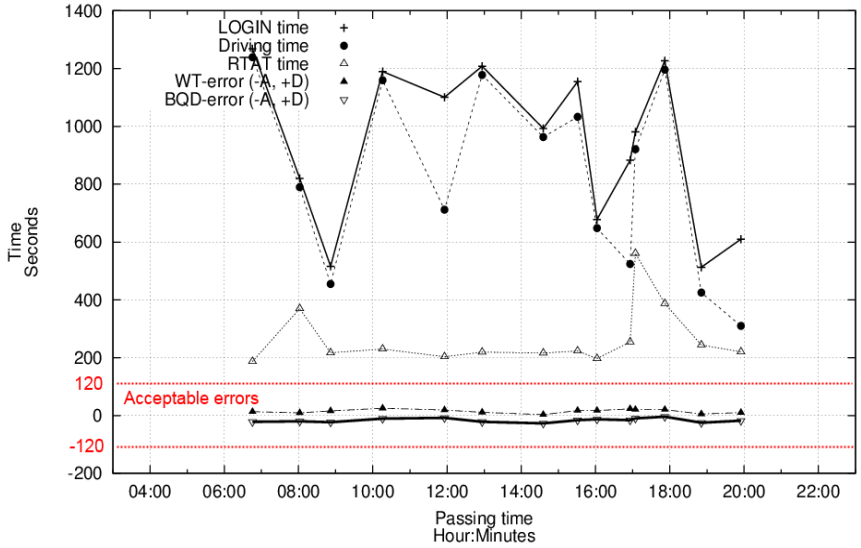

Fig. 6. Example of visual representation of SBK-file (bus line 2132 and monitor 1031)

We can reason a few things in Fig. 6. Times between 14 EPT's have been varied a lot. For example "LOGIN time" in about 8:00 – 9:00 and 16:00 - 17:00 seems to be smaller. These times are typically morning and afternoon rush hours. We see also that real waiting time values have been quite short time (RTAT time: 200s) in monitors, but that is because in this case the trip from the terminal bus stop to the monitor was quite short. Estimation of waiting times are seemed to work quite well, because error values (WT-error and BQD-error) are between +120 - -120 seconds in all EPT cases. These were quite normal results. Additionally traffic jams or other problems can also be easily see from the visual presentation like that.

All detected errors can later be explored statistically from SBK file to get more detailed information of causes of errors. For example bus, line and sign (bus stop) specific errors such as the following error percentages:

BUS,463 = 1 / 24 = 4.2 %
BUS,53 = 1 / 29 = 3.4 %
LINE,2132 DIR 1 = 1 / 42 = 2.4 %
LINE,2132 DIR 2 = 4 / 139 = 2.9 %
SIGN,1033 = 1 / 62 = 1.6 %
SIGN,1061 = 3 / 77 = 3.9 %

These kind of results gave valuable “feedback” knowledge to understand the real behavior of the system. The results also helped to fix bugs and behavior of the system and thus improve the quality of the system.

The execution time of analyzing of one bus line (2132 in this case study) was about 15 seconds using normal PC. There was at all 181 EPT’s in the analysis. Other EPT’s were for other bus stop monitors. The execution time depends on the amount of daily log data and analyzed bus line. There have been three key bus lines and its monitors used for analyzing the ELMI system.
LOGDIG can also be used to detect anomalies, like work flow error or low performance of the execution. For example, the work flow error can be checked by structure of the state machine, and the low performance by time window limits of the ESU’s.

5 Conclusion

We have introduced the LOGDIG analyzer framework that is capable to analyze very complex expected behavior from the logs. LOGDIG was motivated by the fact that there were no other tools flexible enough, even though LFA is very close to the needs. Visualizations and other statistical post-processing have been left out, since there are lots of them available.

LOGDIG is best suited for complex behavior problems, since the setup takes some time compared to e.g. simpler command line search tools. To improve the performance, the Python implementation can be further optimized or even written in compiled language, and the overall execution distributed. If the source log files are available in different machines, the ESUs can be distributed directly there.

In the future, LOGDIG could be used as a higher-level analyzer that uses lower level analyzers. For example, the event search algorithm can be replaced by e.g. LFA/LFA2. To conclude, LOGDIG fulfills the requirements for RTPIS kind of applications, and offers a flexible framework for other domains.

6 References

1. Valdman, J. Log file analysis. Technical report, Department of Computer Science and Engineering, University of West Bohemia in Pilsen (FAV UWB), Czech Republic, , Tech.Rep.DCSE/TR-2001-04 (2001) pp. 1-51.
2. Oliner, A., Ganapathi, A. & Xu, W. Advances and challenges in log analysis. Communications of the ACM 55(2012)2, pp. 55-61.
3. Jayathilake, D. Towards structured log analysis. Computer Science and Software Engineering (JCSSE), 2012 International Joint Conference on, 2012, IEEE. pp. 259-264.
4. Matherson, K. Machine Learning Log File Analysis. Research Proposal (2015).

5. Viklund, J. Analysis of Debug Logs. Master of Science. 2013. Luleå University of Technology, Department of Computer Science, Electrical and Space Engineering, Sweden. 1-39 p.
6. Fu, Q., Lou, J., Wang, Y. & Li, J. Execution anomaly detection in distributed systems through unstructured log analysis. Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on, 2009, IEEE. pp. 149-158.
7. Feather, M.S. Rapid application of lightweight formal methods for consistency analyses. Software Engineering, IEEE Transactions on 24(1998)11, pp. 949-959.
8. Chang, F. & Ren, J. Validating system properties exhibited in execution traces. Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, 2007, ACM. pp. 517-520.
9. Groce, A., Havelund, K. & Smith, M. From scripts to specifications: the evolution of a flight software testing effort. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, 2010, ACM. pp. 129-138.
10. Andrews, J.H. & Zhang, Y. General test result checking with log file analysis. Software Engineering, IEEE Transactions on 29(2003)7, pp. 634-648.
11. Aulenbacher, I.L. Master of Science, Generating Log File Analyzers, The University of Western Ontario, London Ontario Canada (2012) pp. 1-88.
12. LEAL–AULENBACHER, I. & ANDREWS, J.H. Generating C Log File Analyzers. WSEAS Transactions on Information Science & Applications 10(2013)10.
13. Andrews, J.H. & Zhang, Y. Broad-spectrum studies of log file analysis. Proceedings of the 22nd international conference on Software engineering, 2000, ACM. pp. 105-114.
14. Aaltonen, J. Implementation of GPS based real time passenger information system. Licentiate in Technology. 1998. Tampere University of Technology. 1-76 p.
15. Heikkinen, E. Informaatiotaulun protokollakortin ohjelmisto. Master of Science. 1996. Tampere University of Technology. 1-68 p.