AECID-PG: A Tree-Based Log Parser Generator To Enable Log Analysis

Markus Wurzenberger, Max Landauer, Florian Skopik
Austrian Institute of Technology, Center for Digital Safety and Security
Giefinggasse 4, 1210 Vienna, Austria
firstname.lastname@ait.ac.at

Wolfgang Kastner
Vienna University of Technology
Treitlstrasse 3, 1040 Vienna, Austria
k@auto.tuwien.ac.at

***Abstract*—Understanding a computer system's or network's behavior is essential for various tasks such as fault diagnosis, intrusion detection or performance analysis. A key source of information describing a system's current state is log data. However, accessing this information for further analysis is often complicated. Usually, log data is available in form of unstructured text lines and there exists no common standard for the appearance of logs. Hence, log parsers are required to pre-process log lines and structure their information for further analysis. State of the art log parsers still apply pre-defined lists of regular expressions, which are linearly processed and thus render online log analysis infeasible. Furthermore, defining log parsers manually is a cumbersome and time consuming task. Therefore, in this paper we propose AECID-PG, a novel log parser generator. AECID-PG implements a density-based approach to automatically generate a tree-like parser, which reduces the complexity of log parsing from $O(n)$ to $O(log(n))$. We use real log data to evaluate AECID-PG and compare its parsing capabilities to other parser generator approaches by calculating the F-score. We prove AECID-PG's broad applicability and finally demonstrate its functionality in a real world setting.**

***Index Terms*—log data analysis, log data parsing, parser generator**

I. Introduction

Log data is the lowest common denominator of data that any piece of software can produce to inform about its operational state. Thus, log data is a key information source for many different applications such as intrusion detection, fault diagnosis, performance evaluation, predictive maintenance and network behavior analysis. Nowadays, all these techniques are applied in virtually any type of system, being Web-based systems, enterprise IT, cyber physical systems (CPS), Industry 4.0, or the Internet of Things (IoT). However, despite its broad application, there is no common standard for the structure and appearance of log data. As a consequence, it is rather difficult to make data automatically accessible for further analysis with no or minimal manual effort.

Log data occurs in form of unstructured text lines that describe a certain system or network event. Thus, log parsing is an important task prior to log analysis. A log parser knows the syntax, i.e. unique structure, of the data produced by a monitored system or service. Log parsers carry out prepossessing steps to enable further analysis, such as signature and rule verification or anomaly detection. Therefore, parsers sanitize time stamps, disassemble log lines into meaningful tokens, e.g., whit-space separated strings, assign an event type to each line and filter out lines that are irrelevant for further analysis.

978-3-903176-15-7 © 2019 IFIP

However, the following major challenges occur when parsing log data: First, today's modern systems and networks produce large amounts of log data, up to several thousands per second in a medium-sized infrastructure. Thus, parsing log lines must be highly efficient to enable online log analysis, which is especially necessary for critical tasks, including intrusion detection and safety monitoring. Current log parser approaches apply sets of distinct regular expressions to parse log data. This is quite inefficient, with a computational complexity of $O(n)$ per log line, where n is the number of regular expressions. While this is acceptable for forensic analysis, it is not for online processing. Second, each device and network is unique and therefore shows a unique system behavior, because of the users who operate it and the services and applications it runs. Hence, every system needs specific parsers. Furthermore, the complexity of today's networks increases fast and technologies evolve quickly. As a result, also logging infrastructures and the syntax of log lines changes frequently. Consequently, it is a cumbersome and time consuming task to define and maintain log parsers manually. In this paper, we present the following contributions to address these challenges:

(i) A tree-like parser that could be seen as a single very large regular expression that models a system's log data. During parsing a log line, the parser leaves out irrelevant parts of the model and reduces the complexity for log line parsing to $O(log(n))$.
(ii) AECID-PG, a density-based [1] log parser generator approach that automatically builds a tree-like log parser. In opposite to many other parser generator approaches, AECID-PG does not rely on distance metrics. Instead, it uses the frequency with which log line tokens occur.

Since, AECID-PG does not rely on the semantics of the monitored log data, it can be applied in any domain to any log data that has a static syntax. Furthermore, the tree-like structure of the parser allows to address log line parts that include interesting information efficiently, using the relating path of the parser tree. This simplifies accessing information in log lines and speeds up further analysis of the log data, such as rule and signature verification.

The remainder is structured as follows: Sect. II describes the concept of the proposed parser. Next, Sect. III introduces the parser generator AECID-PG, and defines the underlying model to obtain the structure of a considered log file. Sect. IV evaluates AECID-PG, compares it with state of the art log parser generators and includes an application scenario. Finally, Sect. V summarizes related work and Sect. VI concludes the paper.

II. Tree-based Parser Concept

In [2] Wurzenberger et al. conceptually describe a log data parser that leverages a tree-like structure. This kind of parser takes advantage of the inherent structure of log lines.

Developers may freely choose the structure of log lines produced by their services or applications. There are no commonly accepted standards, and industry best practices only define certain aspects of log syntax. For example, the syslog [3] standard dictates that each log line has to start with a time stamp followed by the host name. The rest of the syntax however can be chosen without any restrictions. It is noteworthy that log lines usually consist of static and variable tokens, which are separated by delimiters, such as white spaces, semicolons, equal signs, or brackets.

Applying standards, e.g., syslog, causes log lines produced by the same service or application to be similar in the beginning and differ more towards the end of the lines. Consequently, a parser tree comprises a common trunk and branches towards the leaves, see Fig. 1. The parser tree represents a graph theoretical rooted out-tree. This means, during parsing, it processes log lines token-wise from left to right and only parts of the parser tree that are relevant for the log line at hand are reached. Hence, this type of parser avoids passing over the same log line more than once as would be done when applying distinct regular expressions. As a result, the complexity for parsing reduces from $O(n)$ to $O(log(n))$. Eventually, each log line relates to one path, i.e. branch, of the parser tree.

Figure 1 visualizes a part of a parser tree for ntpd (Network Time Protocol daemon) logs. This example demonstrates that the tree-based parser consists of three main building blocks. The oval orange nodes represent tokens with static text patterns. This means that in all corresponding log lines, a token with this text pattern has to occur at the position of the node in the tree. For example, the first node represents the service name, which in case of syslog data, has to occur in all log lines generated by the ntpd service, after a preamble consisting of a timestamp and the hostname. Pentagonal blue nodes represent nodes that allow variable text until the next separator or static pattern along the path in the tree occurs. For example, the second node relates to the process ID (PID), which is variable and separated by square brackets. The third building block is a branch element. The parser tree branches, when in a certain position only a small number of different tokens with static text occur. This is the case, for example, when a component generates log lines for different events, as in Fig. 1 after the third node. Furthermore, variable nodes can have properties, and allow only specific input, such as numbers or IP-addresses.

Besides the fact that a tree-based parser processes log lines more efficiently than a set of distinct regular expressions, the tree-like structure allows to quickly access information stored in single tokens for further analysis by using the path that addresses the token.

III. AECID-PG: Tree-based Log Parser Generator

AECID-PG implements a density-based parser generator approach, which uses token frequencies instead of a distance metric to determine whether patterns should be static or variable and if a branch element is required. However, the main difference to existing approaches is that this computation is carried out locally in every node of the generated parser tree, rather than for all log lines.

In the remaining section, we will use Fig. 2 to explain our analytical model to generate log parsers. For convenience, the tree represents synthetic log data that includes log lines such as `T D X I Z`, where `T` represents the time stamp of the line. Each line is split into tokens separated by white spaces. The example line would be split into the tokens `T`, `D`, `X`, `I`, `Z`. Assuming that the tokens `X` and `Z` represent variable parts of the log line, the related path of the parser tree includes also variable nodes. Hence, in Fig. 2, letters represent tokens with static and stars tokens with variable patterns.

A. Problem Statement

The simplest method to automatically build a tree-like parser for log data is to use a set of training log lines that represents the normal system behavior and define a tree, where all parts of the log lines are considered static. Therefore, all nodes represent static patterns, and the tree includes all possible unique paths occurring in the log data. Thus, the parser generator creates branches when it registers a sequence of tokens, which is not yet present in the parser tree. For example, in Fig. 2 in column Y_3, the parser generator creates a branch at the top node that represents token `E`. When the parser generator recognized the log line `T A E H J` for the first time, this path represented only log lines with the token sequence `T A E G`. Building a parser tree like that results in a parser, which would perfectly parse the training log data, but if one applies it to other log data, even if it origins from the same system, many log lines would be unparsed, i.e. not reflected accurately by this parser tree. Reasons for this are that (i) unique log line parts, such as IDs and time stamps, and (ii) highly variable parts, such as sensor values, are considered static. Thus, the resulting parser would over-fit the trainings data and could not be practically applied due to its complexity. To avoid an over-fitting parser, AECID-PG applies a set of rules to decide whether it should create a node that represents static text, a node that allows variable text, or a branch into more than one node that represents static text. Also paths that occur too rarely, such as between `E` and `G` in the top of column Y_3 and Y_4 are omitted by the parser generator due to the fact they are outliers and therefore are not part of the normal system behavior.

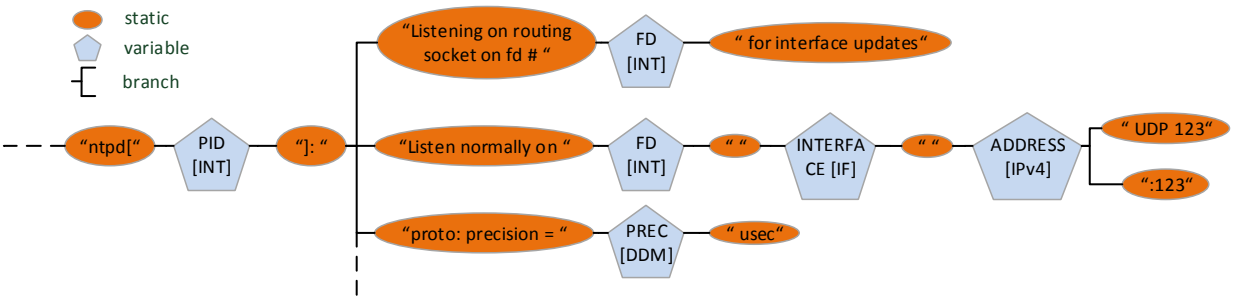

Figure 1. The tree visualizes a part of the parser tree for ntpd (Network Time Protocol) service logs.

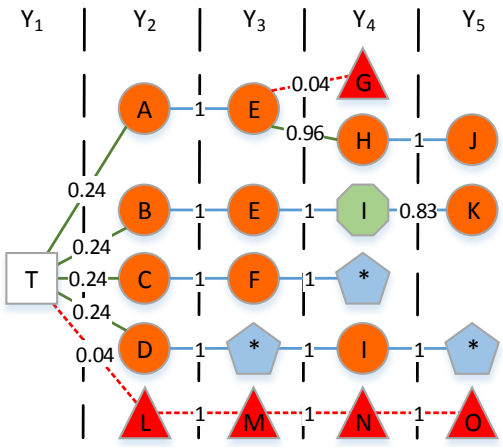

Figure 2. A synthetic parser tree. The square node represents the preamble including the time stamp T, orange circles static nodes, blue pentagons variable nodes, red triangles nodes that occur too rarely to be part of the parser, and green hexagons optional nodes.

B. AECID-PG Concept

Figure 3 visualizes the concept of the AECID-PG approach. In the following, we assume that the parser generator processes the log lines in one batch. The approach basically splits into four steps: (i) Log data is collected. In this paper, we consider textual log data which one or more computer systems or network components produce sequentially in form of log lines. (ii) Each log line is tokenized, i.e. split into meaningful strings. Therefore, a predefined list of delimiters is used that can include symbols such as white spaces, colons, equal signs, brackets, etc. The tokens form the basis to build the parser tree, because they define the nodes of the tree. (iii) The data is transformed into a table, where column Y_i stores a list of the i-th token of the log lines. AECID-PG processes the data column-wise instead of line by line, to improve the runtime of the parser generator. This is faster, because the algorithm applies hash-tables for this purpose and the maximum number of tokens per log line is usually significantly lower than the number of lines the trainings data set consists of. (iv) The algorithm builds the parser tree. Therefore, nodes of tree-depth i correspond to tokens in column Y_i, as also shown in Fig. 2. An edge between two consecutive nodes can only exist, if the corresponding tokens at least once occur consecutively in the same log line. The next section describes how the algorithms decides, which kind of node, i.e., static, branch, variable, etc., it generates.

C. AECID-PG Rules

AECID-PG applies four rules to build a parser tree and to determine the properties of a node. To describe these rules, we define the path-frequency PF_{ij}^k, which describes the frequency by which node n_i^k from column Y_k reaches node n_j^{k+1} in column Y_{k+1} (cf. Eq. (1)), where $|n_i^k|$ defines the number of lines of the trainings set that reached node n_i^k, $k = 0, \ldots, m$ stands for the column number, i.e. tree depth, and $i = 0, \ldots, p$ corresponds with the index of the nodes in column Y_k and $j = 0, \ldots, q$ with the index of the nodes in column Y_{k+1}. We assume that the path-frequency is only calculated between consecutive nodes that are linked with an edge e_{ij}^k, i.e. path.

$$PF_{ij}^k = \frac{|n_j^{k+1}|}{|n_i^k|} \tag{1}$$

In the following, we assume the algorithm builds the parser tree for one column after another, starting with Y_1. All of the following steps are applied to all $n_i^k \in Y_k$, with $i = 0, \ldots, p$. This means also that the following steps are carried out for each node, i.e. the algorithm has only to consider the remaining log lines described by the path of the current node. First, the algorithm applies the previously described simplest approach for the current column Y_k. This means, it keeps all unique tokens as nodes with static patterns. After the initialization of Y_{k+1}, it applies the following rules to refine the tree in the current column. Hence, the algorithm decides whether nodes with static or variable patterns are required, and whether the parser tree needs a branch or not.

Rule 1. When starting from node n_i^k, if there is no node n_j^{k+1}, with existing e_{ij}^k and PF_{ij}^k greater than or equal to θ_1, with $\theta_1 \in [0, 1]$, the algorithm creates a node with a variable pattern, i.e., the parser allows any input (cf. Eq. (2), where VAR stands for a node with a variable pattern).

$$\{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\} = \emptyset \Rightarrow VAR \tag{2}$$

Rule 1 ensures that the algorithm avoids generating nodes with static patterns for tokens that occur rarely in the log data and therefore would lead to an over-fitting parser. In Fig. 2,

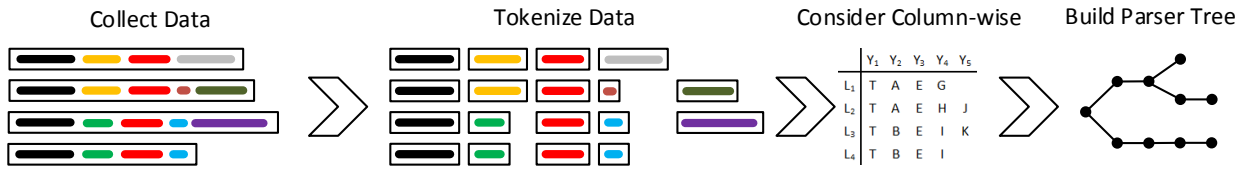

Figure 3. AECID-PG process flow.

this is represented by the blue pentagonal nodes with a star inside.

Rule 2. The second rule is evaluated if there exists exactly one of the generated nodes n_j^{k+1}, with existing e_{ij}^k and PF_{ij}^k greater than or equal to θ_1, i.e. $|\{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}| = 1$. Rule 2 distinguishes the following two cases:

a. If $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$ additionally satisfies Eq. (3), the algorithm generates a single successive node n_j^{k+1} of n_i^k, with a static pattern, that only allows the text of the corresponding token.

$$PF_{ij}^k \geq \theta_2, \text{with } \theta_2 \in [0,1] \tag{3}$$

b. If $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$ does not satisfy Eq. (3), the algorithm creates a node with variable pattern VAR, i.e. the parser allows any input.

Rule 2 ensures that the algorithm does not build a parser model that rejects too many log lines, if the path-frequency to only one node exceeds θ_1, because, for example, if $\theta_1 = 0.1$, the algorithm could reject up to 90% of the log lines that reached the preceding node. Therefore, the path-frequency to this node has to exceed a second higher threshold θ_2. Figure 2 provides an example for Rule 2 in line one between column Y_3 and Y_4. Assuming $\theta_1 = 0.1$ and $\theta_2 = 0.9$, the path-frequency to the upper node `G` does not exceed θ_1 and therefore the node is marked with a red triangle and omitted in the final parser tree. On the other hand, the path-frequency to the lower node `H` exceeds θ_1 and θ_2 and therefore the node is marked with an orange circle and is part of the final parser tree as node representing a static text pattern.

Rule 3. The third rule is evaluated if there exist more than one of the generated nodes n_j^{k+1}, with existing e_{ij}^k and PF_{ij}^k greater than or equal to θ_1, i.e. $|\{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}| > 1$.

Rule 3 distinguishes the following two cases:

a. If $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$ additionally satisfies Eq. (4), where $J = \{j = 0, \ldots, q : n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}\}$ is the set of the indexes of the nodes that satisfy Rule 1, the algorithm generates successive nodes n_j^{k+1} of n_i^k for all $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$, with a static pattern, that only allows the text of the corresponding token.

$$\sum_{j \in J} PF_{ij}^k \geq \theta_3, \text{with } \theta_3 \in [0,1] \tag{4}$$

b. If $n_j^{k+1} \in \{n_j^{k+1} : \exists e_{ij}^k \wedge PF_{ij}^k \geq \theta_1\}$ does not satisfy Eq. (4), the algorithm creates a node with variable pattern VAR, i.e., the parser allows any input.

Similarly to Rule 2, Rule 3 ensures that the algorithm does not build a parser tree that rejects too many log lines. For example, if $\theta_1 = 0.1$, the algorithm could reject up to 80% of the log lines that reached the preceding node, if only 2 nodes have higher path-frequencies than θ_1. Thus, additionally the sum of the path-frequencies to the nodes, which exceed θ_1, has to exceed also a higher threshold θ_3. In Fig. 2, the transition between Y_1 and Y_2 provides an example for Rule 3. Assuming $\theta_1 = 0.1$ and $\theta_3 = 0.95$, the sum of the path-frequencies to the orange circled nodes, representing nodes corresponding to static text patterns, which each exceeds θ_1, exceeds θ_3. If that would not be the case a pentagonal blue node, representing a node corresponding to a variable pattern, would have been generated.

Since, some log lines might end before the path ends, rule 4 is required.

Rule 4. The fourth rule is evaluated, if some log lines end in a node, i.e. before the path ends, and all others succeed. Rule 4 evaluates the following two cases:

a. If the ratio of lines that end in n_i^k is higher than $\theta_4 \in [0,1]$, the algorithm generates all succeeding nodes as optional nodes, i.e. lines can either end before, or reach all succeeding nodes. Otherwise, all lines have to succeed or are considered unparsed.
b. If the ratio of lines that do not end in n_i^k is lower than $\theta_5 \in [0,1]$, the path ends in node n_j^k and there are no succeeding nodes. Otherwise, either rule 4a is true or all lines have to succeed.

Note that θ_4 always has to be greater than or equal to θ_5. In Figure 2, in column Y_4 the top third green octagonal node provides an example for Rule 4. Assuming $\theta_4 = 0.1$ and $\theta_5 = 0.8$, it is possible that optionally some lines end in this node and some exceed it till the end of the path.

D. Features

The remaining section summarizes AECID-PG's most important features. First of all, while most log parser generators only use white-spaces to tokenize the log data, AECID-PG provides the option to freely choose a delimiter and even to define a list of delimiters. Hence, AECID-PG adapts better to log data with different properties and therefore is broadly applicable.

Furthermore, AECID-PG considers path-frequencies locally in each node. Thus, two paths in the parser tree that represent two independent log line classes do not influence each other. Furthermore, it is easier for the parser generator to create branches the farther away the nodes are from the root node. This suits the fact, that log lines are more similar in the beginning than in the end. For example, a syslog line usually starts with time stamp, host name, and in most times service name, before the structure and the content become looser [3].

However, to ensure that the thresholds $\theta_1 - \theta_5$ are globally correct and with increasing tree depth IDs do not become nodes with static patterns, which would make the parser inapplicable, for log data that differs from the training data, AECID-PG includes an optional damping mechanism. The damping mechanism is a function that increases the thresholds θ_i in relation to the current tree depth k, and applies the damping constant Δ (see. Eq (5), where $|n_i^k|$ is the number of lines that reached node n_i^k).

$$\theta_{i_{k+1}} = \theta_{i_k}(1+\Delta), \quad \Delta = 1 - \frac{|n_j^{k+1}|}{|n_i^k|} \tag{5}$$

Moreover, AECID-PG is able to detect predefined patterns, that correspond to the ones the AMiner [2], a log sensor for anomaly detection that leverages a tree-based parser, applies, such as IP addresses, date times, integers, or specified alphabets. These nodes are similar to nodes that allow variable patterns. However, they demand for certain properties of the parsed log line parts.

IV. Evaluation

The following section discusses the evaluation of AECID-PG. The section describes the real world data we used for the calculation of the F-Score[1] and its results, the approaches we compared with AECID-PG and an application scenario. We do not evaluate the performance of AECID-PG, because generating a parser is not a time critical task. Furthermore, the parser approache reduces the complexity of parsing from $O(n)$ to $O(log(n))$ by definition.

A. Experimental data

For the evaluation of AECID-PG, we used five real-world data sets: (i) logs from the supercomputer system BlueGene/L (BGL) [4], (ii) HPC logs from a high performance cluster, which has 49 nodes with 6,152 cores and 128GB memory per node [5], (iii) logs from a 203-node cluster on Amazon EC2 platform (HDFS) [6], (iv) logs from Zookeeper installed on a cluster with 32-nodes [7], and (v) logs form the standalone software Proxifier [7]. Table I describes important properties of the data and demonstrates the complexity of the data sets. The table shows that the data sets are significantly different regarding length of log lines with respect to white-space separated words (excluding time stamps), and number of templates in the ground-truth, which relates to the number of different events logged in the log files (cf. Tab. I).

[1] $F_1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}, Prec. = \frac{TP}{TP+FP}, Rec. = \frac{TP}{TP+FN}$

Table I
Experimental data: number of words per log line, number of templates in the ground-truth, number of templates generated with AECID-PG, AECID-PG input parameters.

	BGL	HPC	HDFS	Zookeeper	Proxifier
Line Length	10-102	6-104	8-29	8-27	10-27
#Templates GT	112	44	14	46	7
#Templ. AECID-PG	120	17	10	17	9
θ_1	0.05	0.05	0.02	0.2	0.05
$\theta_2 = \theta_3$	0.6	0.9	0.9	0.9	0.95
$\theta_4 = \theta_5$	0.01	0.01	0.01	0.01	0.01
Δ	0.1	0.1	0.1	0.01	0.01
Delimiters	‘ ’	‘ ’, =, (,)	‘ ’	‘ ’, @	‘ ’, (,)

B. F-Score Evaluation and comparison with other approaches

For the evaluation of the accuracy of AECID-PG, we calculated the F-Score as shown in [8] and compared AECID-PG to five other parser generator approaches: (i) SLCT (Simple Logfile Clustering Tool), a density-based clustering approach that generates log patterns, (ii) IPLoM (Iterative Partitioning Log Mining) applies a heuristic three-step hierarchical partitioning approach to generate templates, (iii) LKE (Log Key Extraction) that applies clustering and heuristics, (iv) LogSig uses word pair generation and clustering before it generates log templates and (v) Drain, a log parser generator using a fixed depth tree approach [9]. For the choice of input parameters, we oriented us on [9] and [7]. The input parameters of AECID-PG (see Tab. I) depend on the complexity of the input data. For example, a more complex data set requires a smaller θ_1, which makes it easier to generate branches.

For the F-Score evaluation, we randomly chose 2000 lines from each of the log log files described in the previous section. The data, as well as implementations of the aforementioned log parser generator algorithms and their configurations are provided by [7], who also provide a ground-truth for each log file that we leveraged to calculate the F-Score. For the calculation of the F-Score, first, all log lines of the experimental data have been assigned to the correct template of the ground-truth. Then, the parser generators have been applied to the data. Once the parser has been generated, for example in case of AECID-PG, we created a template for each path in the parser-tree, i.e., the path between root node and each leaf node, including optional nodes. Next, we assigned the log lines of the experimental data to the corresponding templates. Finally, we calculated the true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN), as [8] describes: A *TP* decision assigns two lines which are assigned to the same template of the ground-truth also to the same template when considering the templates of the parser generator, a *TN* decision assigns two lines which are assigned to different templates of the ground-truth also to different templates of the parser generator, a *FP* decision assigns two lines which are assigned to different templates of the ground-truth to the same template of the parser generator, and a *FN* decision assigns two lines which are assigned to the same template of the ground-truth to different templates of the parser generator.

Table II summarizes the results of the F-Score evaluation. The input parameters we used for AECID-PG are given in Tab. I. The F-Sore values demonstrate that parsers generated with AECID-PG are either more accurate than the parsers of the compared parser generators or at least comparably accurate. Furthermore, we calculated the average F-Score, which is stored in the last column in Tab. II. AECID-PG achieves the highest average F-Score, which proves its broad applicability.

Table II
COMPARISON OF F-SCORE RESULTS FOR DIFFERENT PARSER GENERATORS AND LOG FILES

	BGL	HPC	HDFS	Zookeeper	Proxifier	Avg
AECID-PG	0.9556	0.9626	0.9996	0.9487	0.8496	0.94322
SLCT	0.6355	0.8109	0.4143	0.8218	0.8707	0.71064
IPLoM	0.9999	0.6485	0.869	0.9995	0.8609	0.87556
LKE	0.4765	0.1793	0.9637	0.8224	0.8684	0.66206
LogSig	0.2653	0.8662	0.9493	0.9906	0.8467	0.783618
Drain	0.9896	0.8576	1	0.9995	0.8609	0.94152

C. Application Scenario

The remaining section describes an application scenario for AECID-PG. Specifically, we took an HDFS log file [6] of around 2 days, consisting of more than 11 million lines. We randomly chose 1% of the lines as training data and used AECID-PG to create a parser tree for HDFS logs. The input parameters we used are: $\theta_1 = 0.005, \theta_2 = \theta_3 = 0.95, \theta_4 = \theta_5 = 0.001, \Delta = 0.01$, and delimiters white-space and underscore. During the training, 0.17% of the training data have not been represented by the resulting parser-tree, because these lines occurred too rarely and therefore have been considered as outliers. We then transformed the parser-tree into a parser for the AMiner[2] and applied it to the whole data set, which only consists of log lines representing normal system behavior. The result was that only 0.23% of the lines were unparsed. Furthermore, the parser processed 84.800 lines per second and it took 132 seconds to process the whole file on a common desktop machine.

V. RELATED WORK

Dealing with massive amounts of log messages is a common problem in log data analysis and requires automatized methods for classifying and parsing log data. A first approach for log clustering using templates was SLCT [1]. The algorithm thereby pursues a density-based, clustering, i.e., frequent words on certain positions in the log lines are considered as fixed, while infrequent words are considered as variables. Distance-based approaches group similar log lines and extract signatures from the resulting clusters. Signatures may then be generated by different approaches, e.g., merging the log messages using string alignments [10], building parse trees based on the number of tokens in the log lines [9], and replacing words that diverge in the grouped sets of log lines with wildcards that represent variable nodes [11]. Another method for generating log signatures is partitioning. Thereby, the groups of log lines are iteratively divided into subgroups by splitting at appropriate token positions, e.g., IPLoM [12]. Finally, recent approaches use neural networks for signature extraction [13].

[2]https://launchpad.net/logdata-anomaly-miner

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented AECID-PG, a novel density-based approach for a parser tree generator for textual computer log data. AECID-PG automatically generates log parsers and therefore reduces time for maintenance of log parsers tremendously. The tree structure of the parser, reduces the computational complexity for log line parsing enormously, in opposite to applying long lists of regular expressions. The so increased performance of log parsing enables online log analysis. The evaluation showed the broad applicability of the AECID-PG approach and demonstrated its functionalities in a real world scenario.

Currently, the parser generator processes the training data as batch, i.e. all log lines at once, and finally provides the tree-like parser. The parser can be applied, for example, with the AMiner a sensor for online anomaly detection [2]. We plan to further develop AECID-PG so that it can also sequentially build the parser tree and adapt the parser according to changes in the system behavior.

ACKNOWLEDGMENT

This work was partly funded by the FFG projects BAESE (852301), synERGY (855457) and INDICAETING (868306).

REFERENCES

[1] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *IPOM 2003*. IEEE, 2003, pp. 119–126.
[2] M. Wurzenberger, F. Skopik, G. Settanni, and R. Fiedler, "AECID: A self-learning anomaly detection approach based on light-weight log parser models," in *Proceedings of the 4th International Conference on Information Systems Security and Privacy, ICISSP 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, 2018, pp. 386–397.
[3] R. Gerhards, "The syslog protocol," Tech. Rep., 2009.
[4] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Dependable Systems and Networks, 2007. DSN'07.* IEEE, 2007, pp. 575–584.
[5] L. LLC., "Operational data to support and enable computer science research." [Online]. Available: http://institutes.lanl.gov/data/fdata
[6] W. e. a. Xu, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 117–132.
[7] L. Team, "Logpai." [Online]. Available: https://github.com/logpai/
[8] C. Manning, P. Raghavan, H. Schütze, and C. U. Press, *Introduction to Information Retrieval*. Cambridge University Press, 2017. [Online]. Available: https://books.google.at/books?id=Sq66tQEACAAJ
[9] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 33–40.
[10] H. e. a. Hamooni, "Logmine: Fast pattern recognition for log analytics," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1573–1582.
[11] K. Shima, "Length matters: Clustering system log messages using length of words," *arXiv preprint arXiv:1611.03213*, 2016.
[12] A. A. Makanju, A. N. Zincir-Heywood, and E. E. Milios, "Clustering event logs using iterative partitioning," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 1255–1264.
[13] V. Menkovski and M. Petkovic, "Towards unsupervised signature extraction of forensic logs," in *Benelearn 2017: Proceedings of the Twenty-Sixth Benelux Conference on Machine Learning, Technische Universiteit Eindhoven, 9-10 June 2017*, 2017, p. 154.