# Abstracting Log Lines to Log Event Types for Mining Software System Logs

Meiyappan Nagappan
*Department of Computer Science*
*North Carolina State University*
*Raleigh, USA*
*Email: mnagapp@ncsu.edu*

Mladen A. Vouk
*Department of Computer Science*
*North Carolina State University*
*Raleigh, USA*
*Email: vouk@ncsu.edu*

***Abstract*—Log files contain valuable information about the execution of a system. This information is often used for debugging, operational profiling, finding anomalies, detecting security threats, measuring performance etc. The log files are usually too big for extracting this valuable information manually, even though manual perusal is still one of the more widely used techniques. Recently a variety of data mining and machine learning algorithms are being used to analyze the information in the log files. A major road block for the efficient use of these algorithms is the inherent variability present in every log line of a log file. Each log line is a combination of a static message type field and a variable parameter field. Even though both these fields are required, the analyses algorithm often requires that these be separated out, in order to find correlations in the repeating log event types. This disentangling of the message and parameter fields to find the event types is called abstraction of log lines. Each log line is abstracted to a unique ID or event type and the dynamic parameter value is extracted to give an insight on the current state of the system. In this paper we present a technique based on a clustering technique used in the Simple Log file Clustering Tool for log file abstraction. This solution is especially useful when we don't have access to the source code of the application or when the lines in the log file do not conform to a rigid structure. We evaluated our implementation on log files from the Virtual Computing Lab, a cloud computer management system at North Carolina State University, and abstracted it to 727 unique event types.**

***Keywords*-log file abstraction; clustering;**

## I. Introduction

Software systems collect information about their activity in log files. The term 'to log' comes from making entries in a logbook to keep track of activities completed. The information in the log files, called logs, consists of the start or end of events or actions of the software system, state information and error information. Each log line typically contains date and time information, user information, application information, and event information. Logs are often collected for system monitoring, system debugging and fault diagnosis. Numerous log file analysis tools and techniques are available to carry out a variety of analyses. Insights of varying degrees are achieved by log file analysis. These include but are not limited to, fault detection by monitoring, fault isolation [5], operational profiling [6] etc. Tools like Splunk [1], and Swatch [3], are used to monitor log files. Splunk is a log management tool, and Swatch is a log monitoring tool.

The users of log files either manually look for a specific piece of information in the log file or apply an analysis algorithm to mine information from it. In the latter case the accuracy of the results produced are highly dependent on the variability present in log files. Each line in a log file is a combination of a static message type and variable parameter information. For eg.

```
Request data from 127.0.0.1 to 127.0.0.2
```

The static fields of the above example are 'Request data from' and 'to'. The parameters in the above example are '127.0.0.1' and '127.0.0.2'. The parameters change at run time and may be different in each instance of the above example in the log file, i.e. the parameter can be any IP address. Hence two instances of the same message will look different because of the different IP address in them. But the same line of code essentially has executed. The separation of the static field from the dynamically changing parameter field is called log file abstraction. Thus in the above example we would separate the log line into two fields:

```
Message Type: Request data from * to *.
Parameter Fields: 127.0.0.1, 127.0.0.2
```

Log file analysis techniques like operational profiling [6], fault isolation [5], system problem detection [9] etc, operate on the abstracted log files. In each of the above techniques, each log line in the log file is abstracted to a corresponding integer ID. Then the analysis is carried on the set of integers. In most of the techniques the abstraction itself is done by regular expressions. The abstraction techniques usually are used to build these regular expressions. The result of abstraction techniques are a set of regular expressions and corresponding IDs. The log lines in the log files can then be matched to these regular expressions and the ID for the best match is used for each line in the log file.

### A. *Contributions*

1) A study of the existing abstraction techniques and their advantages and disadvantages.
2) Our abstraction algorithm which tries to address the issues of the other algorithms and
3) The evaluation of our algorithm

## II. Related Work

A number of techniques are available for log file abstraction. Traditionally the users of log file analysis tools come up with the regular expressions for the abstraction. This can be based on their knowledge of the system, or mining the source code for it [9], or data mining techniques [8] or hybrid of the set of above techniques. One such hybrid technique is proposed by Jiang et al [4]. In this section we discuss in more detail the research by Xu et al. [9], Vaarandi and his tool, SLCT [8], and Jiang et al [4].

Xu et al. [9] in their research on mining console logs for problem detection used a novel technique for the abstraction of the log files. Each unique event type was assigned a unique ID and each line in the log file was abstracted to one of those IDs. Each event type in the set was a regular expression. For example
`starting: xact (.*) is (.*)` [1]
where the "(.*)" is the parameter and the fixed string, "starting: xact" is the message type. This regular expression is assigned a unique ID. A set of these regular expressions are extracted from the source code. Then each line in the log file is compared against each of the regular expressions to find the best match. The ID corresponding to the best match is assigned to this line. Since it is a regular expression they are also able to extract the value of the variable parameter fields in the log lines. These regular expressions are built from the source code. They search the source code for all the calls to the function that prints the message in the log files. This search is done using any standard text editor. The calls to this logging function contain the static message usually withing quotes and have variables for the parameter fields. By replacing the variables with the string "(.*)", they are able to build the regular expressions.

The drawbacks of this solution are that we first need access to the source code. If the people carrying out the analysis are different from the people who develop the application that is logging this information then they may not have access to the source code. Even when we access to the source code there are other issues. In their application Xu et al. had the number of messages under a hundred [9]. But in our prior research [6], where we used a similar abstraction, the number of unique messages were close to 2,000. When there are so many of them spread across multiple files, then it takes a considerable amount of time and manual inspection to build this set of regular expressions. The other issue is that if the message to be printed in the log file is constructed outside the call to the logging function, then we will need to examine the abstract syntax trees to get the regular expressions. Also in some cases the message in the log file is not from the source code of the application. One example is when the source code executes a command on the machine on which the application is executing and prints the output of this command to log file. This log line cannot be abstracted to any of the regular expressions. But if any of the above issues are not present in the log file that we are abstracting, then this solution is the most accurate one.

[1](.*) - Where . = any character, and * = repeated any number of times. Hence (.*) means any character repeated any number of times.

Vaarandi [8] proposes a clustering algorithm for finding patterns in log files. His tool is called the Simple Log file Clustering Tool (SLCT). This algorithm is very similar to apriori algorithms for mining frequent item sets. The modifications made to a typical clustering algorithm for finding frequent item sets, are based on the observations he made about log file data. Firstly only a few words in the log file occur very frequently. Secondly there was a high correlation among these high frequency words. This was because of the fact that each line on the log file is formatted according to the message type and parameter information for that event. This is the same reasoning as in the Xu et al. [9] approach. The message is part of a function call. In his approach Vaarandi does three passes over the log file. The first is to build a data summary, i.e. the frequency count of each word in the log file according to its position in each line. In the second step he builds cluster candidates by choosing log line with words that occur more than the threshold, specified by the user. What is to be noted is that the frequency of a word may be higher in certain positions but may not be so in another position in the log line. In the third step we choose the clusters from these candidates that occur at a frequency higher than the user specified threshold. Each of these candidates are a regular expression. The words in each candidate that have a frequency lower than the threshold are considered as the variable part and hence replaced by "(.*)".

In their research on identifying failure causes, Mariani and Pastore [5], utilize the SLCT to abstract the log lines to log events. But the SLCT was not designed to detect the regular expressions to abstract all lines in a log file. It was designed to detect frequently occurring patterns. A pattern is considered frequently occurring if its frequency is greater than a user specified threshold. All log lines that don't satisfy this condition are stored in the outliers file. Hence even some log lines that repeat a few times but are still less than the threshold won't be considered as a pattern. The goal of that research was to find frequently occurring patterns and not log abstraction. In his paper [8] Vaarandi used thresholds of 50%, 25%, 10 %, 5% and 1%. None of these would either abstract all log lines to an event type or produce regular expressions for abstraction. But if mining frequently occuring patterns is the goal, then this solution performs better than the other ones as it is application and log file independant.

In their research Jiang et al. [4], propose a very efficient approach to log file abstraction. They have 4 steps in their approach: anonymize, tokenize, categorize and reconcile. In anonymize they pick words that they think are parameters.

In their case study they classified the value that followed an '=' or the value following the words 'is—are—was—were'. They replaced these values with a variable '$v'. Then in tokenize they bag log lines with similar characteristics into bins. All log lines in each bin have the same number of static words and same number of parameter fields. Then in the categorize step they go through each bin and compare the anonymized log lines and group all the lines that are exactly the same. Since the variable parts of the line are replaced with a common variable, a simple string comparison would group the log lines. Finally in the reconcile step they go through the groups and combine groups that are different by just one word. This way they are able to group log lines that have a parameter that was not anonymized.

This is a very efficient technique that makes use of the properties of messages in log files for abstraction. But the assumptions are that the logs have enough structure to be able to find the parameters using heuristics. But not all log files have that kind of a rigid structure. The key hurdle when trying to abstract log files is that we don't know what the parameters are in each line of the log file. If we did know, then this technique like the Xu et al. [9], would provide highly accurate results. If log files have some structure to it then highly accurate abstraction can be done. Both Xu et al's [9] and Jiang et al's [4] approach would provide very accurate results when the assumptions under which they operate hold true. When the log files are of a free form and we don't know where the parameters occur and we don't have access to the source code, then only an approximate abstraction solution can be achieved. A clustering algorithm like the SLCT [8] would be such a solution. But as mentioned before, SLCT was not designed to abstract log lines.

In the following section we present our approach, which is a modification of the SLCT algorithm. It makes use of the properties of log files for abstraction.

## III. Our Approach

Vaarandi [8] and Jiang et al. [4] come to the same conclusions about the properties of messages in each line of a log file. We agree with their conclusions and exploit these properties to abstract free form log files. The key property we exploit is that if a particular event occurs in multiple places in a log file with different values for its parameter field, then the static parts of the log line, i.e the words in the message type field will occur many times whereas the variable values will occur fewer times as compared to the static words. We will use the following example to illustrate this.

```
Start processing for Jen user
Start processing for Tom user
Start processing for Henry user
Start processing for Tom user
Start processing for Peter user
```

Table I
Frequency table of the words after Data Summary Step

| **Word** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| Start | 5 | 0 | 0 | 0 | 0 |
| processing | 0 | 5 | 0 | 0 | 0 |
| for | 0 | 0 | 5 | 0 | 0 |
| Jen | 0 | 0 | 0 | 1 | 0 |
| Tom | 0 | 0 | 0 | 2 | 0 |
| ... | 0 | 0 | 0 | 0 | ... |
| user | 0 | 0 | 0 | 0 | 5 |

The words 'Start', 'processing', 'for', 'user' occur 5 times each in the positions 1, 2, 3, and 5 respectively. This is because this is the constant part of the log line. The words 'Jen', 'Henry', and 'Peter' occur once and 'Tom' occurs twice. This is much less than the frequency of other words. From this we can make the inference that the message that created these lines in the log file would be of the following format: 'Start processing for $username user'. This is indeed the statement that created this log message. The words that belong to the constant message type field occur more often than the words in the variable parameter field. We exploit this property to abstract log lines to event types. For this we carry out two passes over the log file.

In the first pass we build a data summary of the words in the log file. We build a frequency table that has the number of times a particular word occurs in a particular position in the log line. Hence the rows in the table are the words, and the columns are the positions in each log line. In Table I we show a part of what the frequency table would look like in the above example after we parse through the 5 lines. Filling the values in this table can be done in time O(N), where N is the number of words in the log file. We take one line at a time and split it into individual words. Then we look up if we have a row for that word in the table. If not we create one. Then we go to this row and increment the value in the column that corresponds to the position of the word in this log line. Looking up the word can be done in constant time by using a hashing function. At the end of the first pass we would have completed building this table.

In the second pass we examine each log line again. Here we do a look up of the table for each word in the line. Then we pick the frequency of that word in that position in the log line from the corresponding column. So for example, when we are parsing through the log line 'Start processing for Jen user', we split it into individual words. Then we look them up in the frequency table. So here we look up, say, 'Start'. Then since 'Start' is the first word in the log line, we retrieve the frequency of the word 'Start' in the first column. Here we would be extracting the value 5. We do this for all the words in the log line.

Then we look for words that occur a similar number of times. Once we find the cluster with the most number of words, we find the lowest of their frequencies. Then any word in the line that has a frequency that is greater than or equal to this value would be a constant word in the log line. In our example, the words ‘Start’, ‘processing’, ‘for’, ‘user’ occur 5 times each. The lowest frequency is 5. Hence all words that are greater than or equal to 5 would be a constant word in the log line. We then save the constant message type and associate it with a unique ID in a list of (message type, ID) pairs. If this is the first occurrence of a message type a new pair is created. Any word that is less than the particular value would be the variable parameters. Hence ‘Jen’ that has a frequency of 1 is a variable parameter. The execution time of this is proportional to the number of words in a line. Hence this step too can be done in O(N) time. So the abstracted version of the log line ‘Start processing for Jen user’ would be ‘1: Jen’, and the list would have the association ‘Start processing for * user: 1’. Hence we have successfully extracted the variable parameter field from the constant message type field. In the case of usernames with two or three words, it might affect the frequency of the word ‘user’ in the example. But in log files we have noticed that there are enough of those (two or three word usernames) log lines too. Hence they too are abstracted the same way to the same message type as well.

In SLCT the author tries to find clusters across log lines [8]. What differentiates our approach from SLCT is that we look for clusters within a log line. The clustering is not of the words but of their frequencies. This is how we are able to identify all the event types in the log file.

## IV. Results

We implemented our approach and tested it on a log file from the Virtual Computing Lab [2], a cloud computing management application at North Carolina State University. The log file was 15 MB big and had 128,636 log lines. Every one of the 128,636 log lines was extracted to one of 727 unique event types. We are able to detect events that occur many hundreds to times to ones that occur just twice. We however cannot detect the message and parameter fields in a log line that has occurred just once. But since it has occurred only once it is unique in itself and hence abstraction is not necessary. Also if the variable value is the same in most of the occurrences of that event, then it will not be detected as one. But the frequency of the variables must be very high for this to happen. SLCT detected only 53 clusters when the threshold was set to 1%. 34,896 log lines were not abstracted at all. This is the disadvantage of SLCT. But then it was designed to detect only the top few clusters, which it found. In the log file that we used in our case study we could not use any of the heuristics given by Jiang et al. [4] or derive new heuristics to consistently find the parameters. Hence we could not perform the anonymize step of their approach on our log file. Therefore we could not compare our technique against theirs. But given a log file where we can effectively anonymize the parameters, their approach would perform better.

## V. Conclusion

Log files contain a lot of information in them and it is often necessary to use an automated analyses technique to mine this information. But the log files have an inherent variability due to the entangling of constant message types and variable parameter types. Hence it is essential for us to abstract the messages in the log file to event types. In the literature there are numerous techniques for log file abstraction. But each of them have their own assumptions like, access to source code or ability to identify parameters using heuristics. These assumptions hold true in a lot of cases and when they do these techniques provide the best and most accurate results. But in the many cases where these assumptions don’t hold true, a more approximate algorithm is required. In this paper we present an approach to log file abstraction that is similar to the SLCT tool. We cluster similar frequency words in each line and abstract it to event types. This is however an approximate approach. As part of future work we intend to calculate the precision and recall of all the four techniques mentioned in this paper on log files from different applications to quantitatively find the better technique under the given conditions.

## Acknowledgment

The authors would like to thank Aaron Peeler and Andy Kurth from the VCL team for providing the log files. This research was funded in part by the DOE grants DE-FC02-ER25809, and DE-AC02-05CH11231.

## References

[1] Splunk http://www.splunk.com/ (accessed 01/12/2010)

[2] Virtual Computing Lab http://vcl.ncsu.edu/ (accessed 01/12/2010)

[3] S.E. Hansen, and E.T. Atkins., Automated System Monitoring and Notification With Swatch. 7th USENIX Conference on System Administration. *System Administration Conference*. Berkeley, CA. November, 1993. pp. 145-152.

[4] Z.M. Jiang, A.E. Hassan, G. Hamann, P. Flora. Abstracting Execution Logs to Execution Events for Enterprise Applications. Journal of Software Maintenance and Evolution:Research and Practice. Volume 20 Issue 4. pp. 249 - 267.

[5] L. Mariani, F. Pastore,. 2008. Automated Identification of Failure Causes in System Logs. 19th International Symposium on Software Reliability Engineering, 10-14 Nov. 2008. pp. 117-126

[6] M. Nagappan, K. Wu, M.A. Vouk,. 2009. Efficiently Extracting Operational Profiles from Execution Logs using Suffix Arrays. 20th International Symposium on Software Reliability Engineering, 16-19 Nov, 2009, Mysuru, India. pp. 41 - 50.

[7] F. Salfner and S.Tschirpke. 2008. Error Log Processing for Accurate Failure Prediction. In Proceedings of the First USENIX Workshop on the Analysis of System Logs, December 7, 2008, San Diego, CA, USA

[8] R. Vaarandi. 2003. A Data Clustering Algorithm for Mining Patterns from Event Logs. IEEE Workshop on IP Operations and Management, 1-3 Oct. 2003. pp. 119-126.

[9] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. 2008. Mining Console Logs for Large-Scale System Problem Detection. SysML‘08, Dec 2008. pp. 1-6.