JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2008; **20**:249–267
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.374

Research

An automated approach for abstracting execution logs to execution events

Zhen Ming Jiang[1,*,†], Ahmed E. Hassan[1], Gilbert Hamann[2] and Parminder Flora[2]

[1]*School of Computing, Queen's University, Kingston, Ont., Canada*
[2]*Enterprise Performance Engineering, Research In Motion (RIM), Waterloo, Ont., Canada*

SUMMARY

Execution logs are generated by output statements that developers insert into the source code. Execution logs are widely available and are helpful in monitoring, remote issue resolution, and system understanding of complex enterprise applications. There are many proposals for standardized log formats such as the W3C and SNMP formats. However, most applications use *ad hoc* non-standardized logging formats. Automated analysis of such logs is complex due to the loosely defined structure and a large non-fixed vocabulary of words. The large volume of logs, produced by enterprise applications, limits the usefulness of manual analysis techniques. Automated techniques are needed to uncover the structure of execution logs. Using the uncovered structure, sophisticated analysis of logs can be performed.

In this paper, we propose a log abstraction technique that recognizes the internal structure of each log line. Using the recovered structure, log lines can be easily summarized and categorized to help comprehend and investigate the complex behavior of large software applications. Our proposed approach handles free-form log lines with minimal requirements on the format of a log line. Through a case study using log files from four enterprise applications, we demonstrate that our approach abstracts log files of different complexities with high precision and recall. Copyright © 2008 John Wiley & Sons, Ltd.

Received 28 December 2007; Revised 30 April 2008; Accepted 7 June 2008

KEY WORDS: execution logs; dynamic analysis; clone detection

1. INTRODUCTION

Execution logs are generated by output statements that developers insert into the source code. Logs record application events at runtime. Logs help in monitoring, remote issue resolution, and system

*Correspondence to: Zhen Ming Jiang, School of Computing, Queen's University, Kingston, Ont., Canada.
†E-mail: zmjiang@cs.queensu.ca

understanding of complex enterprise applications. Tracing and execution logs are two types of logs that are commonly used for dynamic analysis. Tracing logs are generated by instrumenting or monitoring an application using a variety of technologies such as the Java virtual machine profiler interface (JVMPI). In contrast, execution logs are inserted by developers during the development of an application. Execution logs are at a higher level of abstraction than tracing logs. Developers use execution logs to track domain level events (e.g., 'Login Verified'), instead of tracking implementation level events (e.g., 'Function CheckPassword() Called'). There is an abundance of execution logs in the field, whereas tracing logs must be produced for specific scenarios. Producing tracing logs may not be possible in a production setting due to the performance overhead, the lack of system knowledge, and the inaccessibility of the source code. The availability of execution logs continues to increase at a rapid rate due to recent legal acts. For example, the Sarbanes-Oxley Act of 2002 [1] stipulates that the execution of telecommunication and financial applications must be logged.

Execution logs are hard to parse and analyze automatically since they are free-form with no strict format and with a large vocabulary of words. Several standard log formats (e.g., [2–4]) have been proposed to ease the automated analysis of logs. However, such standards are rarely used in practice. Modifying a legacy application to follow a particular standard is usually neither feasible nor economically possible due to the lack of resources, the limited system knowledge, or the inaccessible source code for off-the-shelf applications.

In this paper, we present an approach that can uncover the structure of log lines. The approach examines each log line and abstracts it to its corresponding execution event. Although execution logs may not follow a strict format, they have one general structure: a log line is a mixture of static and dynamic information. Each log line contains static information, indicating the execution event, and dynamic information that is specific to the particular occurrence of the execution event. The dynamic information causes the same execution event to result in different log lines. Looking at Table I, the italic tokens are dynamic information generated at runtime. The rest of a line is static information. Our approach would identify that log lines in Table I correspond to two execution events: 'User checkout' and 'Item shipped'. For example, the first log line would be abstracted to the 'User checkout' execution event.

Once a log file is processed and the execution event corresponding to each log line is identified, a developer can use the recovered log file structure to understand and investigate the dynamic behavior of a software application. Our abstraction approach reduces the volume of data that a developer needs to investigate. Moreover, the abstracted events can be used to reason about important events in a log file. Depending on the type of information recorded and the logging details, different types of analysis can be conducted. For example, the log file of one application in our case study (see Section 5) contains more than 1.6 million log lines. More than 23 000 lines in this log file contain the word 'fail' or 'failure'. A developer investigating these failures would need to manually go

Table I. Example log lines.

1.	User checkout for accountId(*tom*), item = *100*
2.	User checkout for accountId(*jerry*), item = *100*
3.	Item shipped for accountId(*tom*), item = *100*
4.	User checkout for accountId(*john*), item = *103*

DOI: 10.1002/smr

through each one of these lines. Alternatively, our approach abstracts these 1.6 million log lines to 319 execution events. Among these 319 events, there are 12 types of failure events that range from external protocol communication failures to internal synchronization failures. Using the frequency of failure events, the developer can prioritize his work by tackling the most frequently occurring failures first.

1.1. Organization of the paper

The paper is organized as follows. Section 2 overviews related work in the field and places our contributions relative to prior work. Section 3 gives an overview of the process for abstracting log lines. We use precision and recall, two traditional information retrieval metrics, to measure the performance of log abstraction approaches. Section 4 presents our approach of abstracting log lines to execution events. Section 5 demonstrates the effectiveness of our approach through a case study using log files from four enterprise software applications. We also discuss lessons learned from our study. Section 6 concludes the paper.

2. RELATED WORK

Uncovering the structure of free-form text is commonly referred to as the grammar inference problem [5,6]. Prior approach for inferring the grammar of execution logs (i.e., abstracting log lines to execution events) could be grouped under three general approaches: *Rule-based*, *Codebook-based*, and *AI-based* approaches.

Rule-based approaches [7–10] use a set of hard-coded rules for abstracting log lines to execution events. These approaches are commonly used in practice since they are very accurate. However, these approaches require a substantial effort for encoding and updating the rules. For the logs shown in Table I, a rule-based approach would define two regular expressions to map each log line to one of the two possible execution events: the 'User checkout' or 'Item shipped' events.

Codebook-based approaches [11–13] are similar to the rule-based approach. However, codebook approaches process a subset of execution events ('alarms') instead of all events. The subset of events, which forms the codebook, is used in real time to match the observed symptoms. For the logs shown in Table I, a codebook-based approach may consider only tracking the 'Item shipped' events so an alarm for that event would be created using a regular expression.

AI-based approaches [14–21] use various types of artificial intelligent techniques, such as Bayesian networks, frequent-itemset mining, to abstract execution logs to execution events. For the logs shown in Table I, a frequent-itemset approach would recognize the high repetition of the 'User checkout' event. However, the approach would not recognize the 'Item shipped' event since it does not occur that frequently.

Our approach, presented in Section 4.2, is a mixture of rule-based and AI-based approaches. Our approach requires less system knowledge and effort than other approaches. Rather than encoding rules to recognize specific execution events, our approach uses a few general heuristics to recognize static and dynamic information in log lines. Log lines with identical static information are then grouped together to abstract log lines to execution events.

DOI: 10.1002/smr

Table II. Summary of related work.

Approach	Interpretability	Needed knowledge	Required effort	Coverage
Rule-based [7–10]	Y	High	High	N
Codebook-based [11–13]	Y	Medium	High	N
AI-based [14–21]	N	Low	Low	N
Our approach	Y	Low	Low	Y

We define several criteria (Table II) to summarize the difference between these four log abstraction approaches.

1. *Interpretability*: Whether a user can easily understand the rationale for abstracting a log line to a particular execution event? For example, in a neural-network AI approach, the user cannot determine the rationale for abstracting a log line to a particular event. We desire an approach with high transparency so users would trust it and adopt it.
2. *Needed system knowledge*: What is the amount of knowledge needed about the system for the approach to work? For example, in a rule-based approach a domain expert is needed to encode all the rules.
3. *Required effort*: What is the amount of effort required for the approach to work properly? Rule-based and cookbook-based approaches require a large amount of human effort to encode the rules or alarms. These encodings must be updated for every version of a software system.
4. *Coverage*: Is each log line abstracted to an appropriate execution event? For example, some AI approaches only abstract log lines, which occur above a particular threshold.

3. MEASURING THE PERFORMANCE OF APPROACHES FOR LOG ABSTRACTION

Given the four log lines shown in Table I, a log abstraction approach would determine that log lines 1, 2, and 4 correspond to the execution event: 'User checkout for accountId($v), item = $v', where $v indicates the dynamic parts of an execution event. Log line 3 corresponds to another execution event, namely 'Item shipped for accountId($v), item = $v'. Owing to the simple structure of the log lines used in our example, a log abstraction approach could easily perform the abstraction. However, in practice the structure of log lines is not as simple and the performance of each approaches differs between applications.

We use precision and recall, two traditional information retrieval metrics, to measure the performance of different approaches for abstracting log lines to execution events. We first measure the

DOI: 10.1002/smr

performance of an approach for each execution event, then we sum up the performance for each log line to determine the overall performance of the approach. Given a single execution event (e), we know that log lines (A, B, C, F) correspond to e (see Figure 1). On the other hand, the log abstraction approach abstracted log lines (A, B, C, D, E) to event e. Therefore, the log abstraction approach correctly classified log lines A, B, C, incorrectly classified log lines D, E, and missed classifying event F. For event e, we define the number of log lines classified correctly as PC_e, the number of lines classified incorrectly as PF_e, the number of missed lines as PM_e. Using the information from Figure 1, we have for event e:

$$PC_e = \{A, B, C\}, \quad PF_e = \{D, E\} \quad \text{and} \quad PM_e = \{F\}$$

We define precision and recall as

$$precision = \frac{PC_e}{PC_e + PF_e} \quad \text{and} \quad recall = \frac{PC_e}{PC_e + PM_e}$$

Therefore, the approach used in our example would have a recall of $3/(3+2)$ or 60%, and a precision of $3/(3+1)$ or 75%.

In the ideal situation, PF_e and PM_e are empty and then both precision and recall would reach their maximum value. The maximum value for precision and recall is 1. We desire an approach with high precision and high recall.

3.1. Average performance

The above formulas measure performance for a particular execution event (e). To measure the overall performance of an approach, we define the average precision and the average recall, which combine the precision and recall measures for all unique k events ($e_1, e_2, \ldots, e_k$) in a log file as follows:

$$Average\ precision = \frac{1}{k} \times \sum_{1}^{k} precision_{e_i} \quad \text{and} \quad Average\ recall = \frac{1}{k} \times \sum_{1}^{k} recall_{e_i}$$

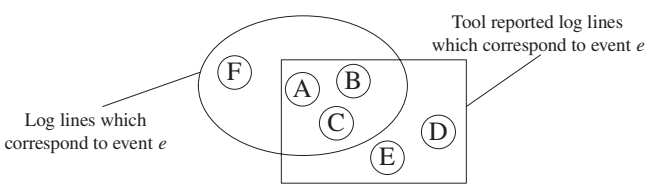


Figure 1. Measuring the performance of a log abstraction approach.

DOI: 10.1002/smr

We use the average precision and recall measures in the remainder of the paper to compare log abstraction approaches. We desire an approach that maximizes precision and recall to reduce the need for manual analysis of log lines.

4. OUR LOG ABSTRACTION APPROACH

In this section, we present our log abstraction approach. Our approach uses clone detection techniques to uncover common tokens in log lines and to parameterize each log line. We first report on our experience in using an off-the-shelf clone detection tool to perform the log abstraction and then present our approach in detail.

4.1. Clone detection

Log lines generated due to the same execution event tend to have high textual similarities (see lines 1, 2, and 4 in Table I). The process of abstracting log lines to execution events can be considered as detecting and grouping similarities among log lines. This intuition leads us to consider using clone detection approaches for abstracting log lines to events.

Code clones refer to identical or similar segments of source code. Code clones are created by copy-and-paste practices adopted by developers to reuse certain design patterns or to minimize risks [22]. Figure 2 shows a code clone example taken from the driver code of the Linux kernel version 2.6.16.13. The example shows three code snippets. The source of each snippet is shown at the top of the figure. The cloned areas are shown in gray with the variation points in bold italic.

Clone detection approaches uncover clones using different measures of similarity. For example, some approaches compare the abstract syntax tree (AST) for two code segments [23], whereas other approaches compare a set of data flow and control metrics [24] or measure the text similarities of

linux-2.6.16.13\drivers\net\ne2.c : 285 - 295

```
#ifndef MODULE
struct net_device * __init ne2_probe(int unit)
{
        struct net_device *dev = alloc_ei_netdev();
        int err;

        if (!dev)
                return ERR_PTR(-ENOMEM);

        sprintf(dev->name, "eth%d", unit);
        netdev_boot_setup_check(dev);

        err = do_ne2_probe(dev);
        if (err)
                goto out;
        return dev;
out:
        free_netdev(dev);
        return ERR_PTR(err);
}
#endif
```

(A)

linux-2.6.16.13\drivers\net\lne390.c: 152 - 162

```
#ifndef MODULE
struct net_device * __init lne390_probe(int unit)
{
        struct net_device *dev = alloc_ei_netdev();
        int err;

        if (!dev)
                return ERR_PTR(-ENOMEM);

        sprintf(dev->name, "eth%d", unit);
        netdev_boot_setup_check(dev);

        err = do_lne390_probe(dev);
        if (err)
                goto out;
        return dev;
out:
        free_netdev(dev);
        return ERR_PTR(err);
}
#endif
```

(B)

linux-2.6.16.13\drivers\net\lance.c: 437 - 447

```
#ifndef MODULE
struct net_device * __init lance_probe(int unit)
{
        struct net_device *dev = alloc_etherdev(0);
        int err;

        if (!dev)
                return ERR_PTR(-ENODEV);

        sprintf(dev->name, "eth%d", unit);
        netdev_boot_setup_check(dev);

        err = do_lance_probe(dev);
        if (err)
                goto out;
        return dev;
out:
        free_netdev(dev);
        return ERR_PTR(err);
}
#endif
```

(C)

Figure 2. Clone example taken from Linux kernel version 2.6.16.13.

DOI: 10.1002/smr

code segments and report code segments as clones if the measurement of similarities exceeds a threshold [25].

We expect a clone detection approach to determine that lines 1, 2, and 4 in Table I are clones of each other with slight differences due to parameterization (i.e., *tom* vs *jerry* vs *john* and *100* vs *100* vs *103*). To verify our intuition, we experimented with using the Kamiya *et al.* algorithm to abstract logs. We experimented with Kamiya *et al.*'s CCFinder [26] tool to abstract log lines. CCFinder, which uses a parameterized token matching algorithm, detects similarities in multiple programming languages and plain text. The tool is easy to use and is fully automated. The only input required is the name of the files to be analyzed, their type, and number of similar tokens. The input files could be C, C++, COBOL, Java, or plain text. The number of similar tokens is a user-specified threshold that is used to determine whether two code segments are similar. If the similarities between two code segments exceed this threshold, they will be reported as clones. The tool scales well to handle large software systems, such as Apache, FreeBSD, NetBSD, and Linux [26–29], with thousands or millions of lines of code.

After running CCFinder on a log file, we discovered that CCFinder is not suitable for our purposes. In particular, CCFinder has the following limitations:

1. *Threshold*: CCFinder has a configuration parameter that specifies the minimum number of tokens that two segments of code should have in common to qualify as a clone pair. However, each execution event contains a different number of words. Therefore, it is not possible to give a single threshold number for all execution events. A large threshold would only process execution event with a large number of tokens and ignore execution events with a small number of tokens. A large threshold will increase the precision of the approach and decrease its recall. Conversely, a small threshold will increase the recall but decrease the precision. A percentage threshold would be more suitable for our purposes.
2. *File size*: CCFinder could only handle small-sized log files. Large files resulted in CCFinder crashing due to excessive use of memory. We require a tool that can process log files with thousands or millions of lines.
3. *Delimiters*: CCFinder performs clone detection across multiple lines. However, this is not ideal for abstracting log lines, since we want the output to stop at the line boundary.

Although CCFinder works well on large source code bases, it is not able to process large log files. We believe this is due to the following reasons:

1. Source code and plaintext wrap around lines but have delimiters for each statement (such as ';', '.', or '!'); whereas a log line does not use similar delimiters. Therefore, CCFinder cannot find the end of each log line and treats all log lines as one large chunk.
2. Source code contains control keywords such as if, else, for, while, etc. These keywords are the static parts in the source code and are used by CCFinder to mark the static parts of the source code. As log lines have a less strict grammar and a non-fixed vocabulary; CCFinder cannot mark up any specific parts as static when processing log lines.

4.2. The steps of our approach

Based on our experience using CCFinder, we implemented our own tool to detect similarities among log lines, and then to parameterize and abstract log lines. Our approach addresses the following

DOI: 10.1002/smr

problems:

1. *Threshold*: Our approach minimizes the impact of threshold for deciding whether two log lines are similar to each other.
2. *File size*: Our approach scales up to process log files that contain thousands or millions of log lines.
3. *Delimiters*: Our approach uses flexible heuristics to mark the dynamic and static information for each log line. We treat end of line characters as the delimiter for each log line.

As shown in Figure 3, our approach consists of four steps: anonymize, tokenize, categorize, and reconcile. In the remainder of this section, we demonstrate our approach using a small running example that is shown in Table III. The example has five log lines.

4.2.1. The anonymize step

The anonymize step uses heuristics to recognize dynamic tokens in log lines. Once dynamic tokens are recognized, they are replaced with a generic token ($v). Heuristics can be added or removed from this step based on domain knowledge. The following are two heuristics to recognize dynamic parts in log lines:

1. Assignment pairs like '*word*=*value*';
2. Phrases like '*is*[*are*|*was*|*were*] *value*'.

Table IV shows the sample log lines after the anonymize step. In the second and third lines, tokens after the equal signs (=) are replaced with the generic term $v. In the fourth line, the phrase 'is 250' is replaced with the term '=$v'. There are no changes made to the first and last lines.

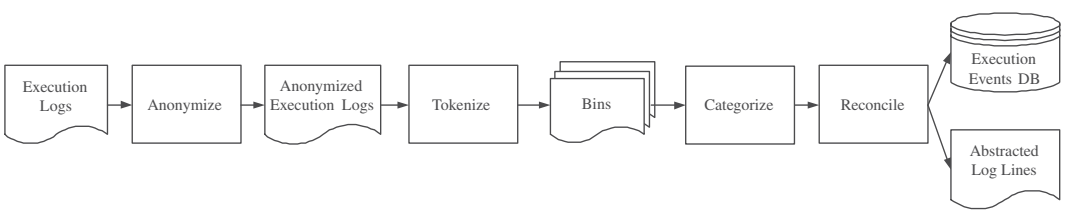


Figure 3. High-level overview of our approach for abstracting execution logs to execution events.

Table III. Log lines used as a running example to explain our approach.

1.	Start check out
2.	Paid for, item=bag, quality=1, amount=100
3.	Paid for, item=book, quality=3, amount=150
4.	Check out, total amount is 250
5.	Check out done

DOI: 10.1002/smr

Table IV. Running example logs after the anonymize step.

1.	Start check out
2.	Paid for, item = $v, quality = $v, amount = $v
3.	Paid for, item = $v, quality = $v, amount = $v
4.	Check out, total amount = $v
5.	Check out done

Table V. Running example logs after the tokenize step.

Bin names (no. of words, no. of parameters)	Log lines
(3, 0)	1. Start check out 5. Check out done
(5, 1)	4. Check out, total amount = $v
(8, 3)	2. Paid for, item = $v, quality = $v, amount = $v 3. Paid for, item = $v, quality = $v, amount = $v

4.2.2. The tokenize step

The tokenize step separates the anonymized log lines into different groups (i.e., bins) according to the number of words and estimated parameters in each log line. The use of multiple bins limits the search space of the following step (i.e., the categorize step). The use of bins permits us to process large log files in a timely fashion using a limited memory footprint since the analysis is done per bin instead of having to load up all the lines in the log file. We estimate the number of parameters in a log line by counting the number of generic terms (i.e., $v). Log lines with the same number of tokens and parameters are placed in the same bin.

Table V shows the sample log lines after the anonymize and tokenize steps. The left column indicates the name of a bin. Each bin is named with a tuple: number of words and number of parameters that are contained in the log line associated with that bin. The right column in Table VI shows the log lines. Each row shows the bin and its corresponding log lines. The second and the third log lines contain 8 words and are likely to contain 3 parameters. Thus, the second and third log lines are grouped together in the (8, 3) bin. Similarly, the first and last log lines are grouped together in the (3, 0) bin since they both contain 3 words and are likely to contain no parameters.

4.2.3. The categorize step

The categorize step compares log lines in each bin and abstracts them to the corresponding execution events. The inferred execution events are stored in an execution events database for future references. The algorithm used in the categorize step is shown below. Our algorithm goes through the log lines

DOI: 10.1002/smr

Table VI. Running example logs after the categorize step.

Execution events (word_parameter_id)	Log lines
3_0_1	1. Start check out
3_0_2	5. Check out done
5_1_1	4. Check out, total amount = $v
8_3_1	2. Paid for, item = $v, quality = $v, amount = $v
8_3_1	3. Paid for, item = $v, quality = $v, amount = $v

bin by bin. After this step, each log line should be abstracted to an execution event. Table VI shows the results of our working example after the categorize step.

```
for each bin b_i
 for each log line l_k in bin b_i
  for each execution event e_(b_i,j) corresponding to b_i in the events
 DB
   perform word by word comparison between e_(b_i,j) and l_k
   if (there is no difference) then
    l_k is of type e_(b_i,j)
    break
   end if
  end for // advance to next e_(b_i,j)
  if ( l_k does not have a matching execution event) then
   l_k is a new execution event
   store an abstracted l_k into the execution events DB
  end if
 end for // advance to the next log line
end for // advance to the next bin
```

We now explain our algorithm using the running example. Our algorithm starts with the $(3, 0)$ bin. Initially, there are no execution events that correspond to this bin yet. Therefore, the execution event corresponding to the first log line becomes the first execution event namely 3_0_1. The _1 at the end of 3_0_1 indicates that this is the first execution event to correspond to the bin, which has 3 words and no parameters (i.e., bin 3_0). Then the algorithm moves to the next log line in the $(3, 0)$ bin, which contains the fifth log line. The algorithm compares the fifth log line with all the existing execution events in the $(3, 0)$ bin. Currently, there is only one execution event: 3_0_1. As the fifth log line is not similar to the 3_0_1 execution event, we create a new execution event 3_0_2 for the fifth log line. With all the log lines in the $(3, 0)$ bin processed, we can move on to the $(5, 1)$ bin. As there are no execution events that correspond to the $(5, 1)$ bin initially, the fourth log line gets assigned to a new execution event 5_1_1. Finally, we move on to the $(8, 3)$ bin. First, the second log line gets assigned with a new execution event 8_3_1 since there are no execution events corresponding to this bin yet. As the third log line is the same as the second log line (after the anonymize step), the third log line is categorized as the same execution event as the second log

DOI: 10.1002/smr

line. Table VI shows the sample log lines after the categorize step. The left column is the abstracted execution event. The right column shows the line number together with the corresponding log lines.

4.2.4. The reconcile step

Since the anonymize step uses heuristics to identify dynamic information in a log line, there is a chance that we might miss to anonymize some dynamic information. The missed dynamic information will result in the abstraction of several log lines to several execution events that are very similar. Table VII shows an example of dynamic information that was missed by the anonymize step. The table shows five different execution events. However, the user names after 'for user' are dynamic information and should have been replaced by the generic token '$v'. All the log lines shown in Table VII should have been abstracted to the same execution event after the categorize step. The reconcile step addresses this situation. All execution events are re-examined to identify which ones are to be merged. Execution events are merged if:

1. They belong to the same bin.
2. They differ from each other by one token at the same positions.
3. There exists a few of such execution events. We used a threshold of five events in our case studies. Other values are possibly based on the content of the analyzed log files. The threshold prevents the merging of similar yet different execution events, such as 'Start processing' and 'Stop processing', which should not be merged.

Looking at the execution events in Table VII, we note that they all belong to the '5_0' bin and differ from each other only in the last token. Since there are five of such events, we merged them into one event. Table VIII shows the execution events from Table VII after the reconcile step. Note that if the '5_0' bin contains another execution event: 'Stop processing for user John'; it will not be merged with the above execution events since it differs by two tokens instead of only the last token.

Table VII. Sample logs that the categorize step would fail to abstract.

Event IDs	Execution events
5_0_1	Start processing for user Jen
5_0_2	Start processing for user Tom
5_0_3	Start processing for user Henry
5_0_4	Start processing for user Jack
5_0_5	Start processing for user Peter

Table VIII. Sample logs after the reconcile step.

Event IDs	Execution events
5_0_1	Start processing for user $v

DOI: 10.1002/smr

5. CASE STUDY

We conducted a case study to evaluate the effectiveness of our approach in abstracting the logs for enterprise applications. We used log files from four different applications: App 1, App 2, LoadSim, and Blue Gene/L. App 1 is a large-scale enterprise application developed by research in motion (RIM). App 2 is a medium-scale enterprise application developed by RIM. Loadsim is a medium-scale enterprise application developed by Microsoft. These three applications are deployed and used by millions of users in thousands of enterprises worldwide. Blue Gene/L logs [30] are from an application running on the Blue Gene/L supercomputer [31]. Table IX tabulates the applications and the size of studied log files. Table X shows sample log lines taken from LoadSim and Blue Gene/L.

5.1. Other approach for log abstraction

In the case study, we want to compare the performance of our approach with other approaches for log abstraction. Since we have limited knowledge of the studied software applications, we could not use rule-based or codebook-based approach. We could only use an AI-based approach. There exist two AI tools against which we could compare our approach. The tools are: teirify [19] and Simple Logfile Clustering Tool (SLCT) [20]. The teirify tool uses a bio-informatics algorithm [32] to detect line patterns, whereas the SLCT uses frequent-itemset mining techniques to cluster similar log lines. Unfortunately, teirify requires a large amount of memory and cannot handle log files, exceeding 10 000 log lines. In the case study, we compared the performance of our approach against the result obtained from SLCT, which scaled to handle large files. We ran SLCT with the $-s$ ⟨*support threshold*⟩ and $-j$ options. The $-s$ ⟨*support threshold*⟩ option specifies a support threshold value. Each line pattern exceeding this threshold will be outputted. For each outputted line pattern, SLCT also shows the support count associated with this pattern, i.e., the number of input lines that correspond to this pattern. Without the $-j$ option, a log line will only be counted

Table IX. Size of the log files of the four studied applications.

Application	App 1	App 2	LoadSim	Blue Gene/L
Number of log lines	723 608	1 688 876	67 651	2 994 986

Table X. Sample log lines from LoadSim and Blue Gene/L.

Sample LoadSim logs	Sample Blue Gene/L logs
Browse Mail: Read 1, Deleted 1	RAS KERNEL INFO program interrupt
Sent oups4k.msg (4 recipient(s))	RAS KERNEL INFO generating core.406
Browse Mail: Read 1, Moved 1	RAS KERNEL INFO program interrupt
There are 6 rules: Added 1 rule	RAS KERNEL FATAL data TLB error interrupt
Sent oups2k.msg (3 recipient(s))	RAS KERNEL FATAL data TLB error interrupt

DOI: 10.1002/smr

toward the support count of a single pattern. With the $-j$ option specified, if a log line matches multiple line patterns then this line will be counted multiple times for the support count for each matched line patterns. Table XI shows an example of the SLCT output. The left column shows five sample input log lines. The right column shows the results of several runs for SLCT using these input log lines. For each run, the table shows the used SLCT options and the corresponding output. If the support threshold is 4, the 'In Checkout, user is *' pattern is outputted. All five input lines match the outputted pattern so the support for this pattern is 5. If the support threshold is 3, the 'In Checkout, user is *' and 'In Checkout, user is Tom' patterns are outputted. The 'In Checkout, user is Tom' is a sub-pattern of 'In Checkout, user is *', so all lines that are mapped to the 'In Checkout, user is Tom' pattern are mapped to the 'In Checkout, user is *' pattern as well. If the support threshold is 2, the 'In Checkout, user is Tom' and 'In Checkout, user is Jerry' patterns are outputted.

For our case study, we wrote a *Perl* script that processes the SLCT output to ease our comparison process. Since a long line can correspond to several line patterns, our script matches each log line to the pattern with the largest support value. For example, using the output from '$-s\ 3\ -j$', the log line 'In Checkout, user is Tom' will be matched with the 'In Checkout, user is *' pattern. We also explored mapping a line to the pattern with the smallest support value. The precision values using the smallest support mapping are similar to the large support mapping. However, the recall values are slightly lower when mapping to the smallest support value.

5.2. Measuring the performance of an approach

To measure the performance of a log abstraction approach, we need to know the correct mapping between every log line and its corresponding execution event. Acquiring such mapping is challenging, even if the source code of an application is available. Simply searching for 'LOG' statements (e.g., 'printf' statements in C/C++ or 'System.out' statements in Java) is not sufficient since in many instances an execution event may be generated dynamically using several output statements in the source code. For example, the log line, shown in Figure 4, is generated by three output statements: one statement is outside the loop and another two statements are inside the loop.

For our case study, we used two techniques to determine the gold standard (i.e., the accurate mapping of each log line to its corresponding execution event). For application App 1, we used an internationalization file to determine the mappings. The application was internationalized and part of the internationalization efforts involved the manual mapping of each log line to an execution event. The execution events are stored in a separate file that is translated to different languages.

Table XI. An example of SLCT output.

	SLCT runs	
Log lines	Options	SLCT output
In Checkout, user is Tom	$-s\ 4\ -j$	In Checkout, user is * (5)
In Checkout, user is Jerry	$-s\ 3\ -j$	In Checkout, user is * (5)
In Checkout, user is Tom		In Checkout, user is Tom (3)
In Checkout, user is Tom	$-s\ 2\ -j$	In Checkout, user is Tom (3)
In Checkout, user is Jerry		In Checkout, user is Jerry (2)

DOI: 10.1002/smr

Log Lines	Source Code

User Shopping Basket contains: 2, 3, 5	LOG("User Shopping Basket contains: ");
	for (int i=0; i<shoppingBasket.size(); i++)
	{
	itemId = shoppingBasket[i];
	if(i > 0) {
	LOG(" , " + itemId);
	} else {
	LOG(itemId);
	}
	}

Figure 4. An example of an execution event generated by multiple output statements.

Table XII. Performance of both approaches on the studied applications.

Application	SLCT precision (%)	SLCT recall (%)	Our precision (%)	Our recall (%)
App 1	31.7	13.4	100	92.6
App 2	23.1±8.2	7.5±8.2	84.2±8.2	82.4±8.2
LoadSim	9.1±8.2	9.1±8.2	87.9±8.2	85.3±8.2
Blue Gene	33.3±8.2	28.3±8.2	100±8.2	100±8.2

This file acted as the gold standard in our performance evaluation for App 1. For the other three applications (App 2, LoadSim, and Blue Gene/L), we performed random sampling to measure the performance of the log abstraction approaches. We randomly picked 100 log lines and measured the performance of SLCT and our approach on these log lines. Such sample size is sufficient to ensure a confidence level of 90% and a confidence interval of ±8.2% for the measured precision and recall results. We believe that a standard corpus, based on the logs of several enterprise applications, would be very valuable for studying and comparing the performance of log abstraction approaches. For this work, our random sampling and the internationalization file were the only possible options instead of a complete manual analysis of the large log files.

Table XII tabulates the performance results for the two approaches. We note that since SLCT uses a frequent-itemset technique, it requires a minimum support count as a parameter. We experimented with different support counts: 10, 50, 100, 150, 200, 250, 500, 750 and 1000. Owing to page limitations, we do not discuss the details of these experiments. However, our experiments show that the differences in performance due to different support counts are not statistically significant (within 2% for recall and within 1% for precision) using an ANOVA analysis with an alpha of 0.05. For reported results, we use a support count of 100.

The results reported in Table XII show that our approach abstracts log lines to events with high precision and recall across the studied applications. On the other hand, the performance of SLCT is not as high. The low precision and recall values for SLCT are due to the following two reasons.

1. Many log lines are not abstracted to any execution event by SLCT since these lines do not occur often enough for a frequent pattern to emerge.
2. Although SLCT outputs a higher support count for the more general line pattern with the $-j$ option, it does not attempt to abstract patterns further. For example, as shown in Table XI for

DOI: 10.1002/smr

support count 2, SLCT would output these two similar patterns 'In Checkout, user is Tom' and 'In Checkout, user is Jerry'. SLCT does not attempt to merge and abstract identified patterns further since it may lead to all patterns being abstracted to a very general '*' pattern.

Our approach does not suffer from the problem of limited frequencies and subpattern merging since we map log lines to events even if the line occurs a single time and our reconcile step identifies similar yet different by one token patterns (i.e., events) and merges them.

5.2.1. Adjusting our log abstraction heuristics

Our approach uses heuristics to recognize the dynamic information in a log line. These heuristics are based on coding conventions and observations made by examining log lines. However, these heuristics are neither necessarily complete nor applicable across applications. For different software systems, even different versions of a system, we might need to adjust our heuristics accordingly:

1. The anonymization rules depend on the application and might need to be adjusted for each application. For the Blue Gene/L logs, the parameter values are printed in 'name:value' style rather than the 'name = value' style. For both App 1 and App 2 logs, we need to anonymize email addresses. For the LoadSim logs, we need to anonymize different message file names ('*.msg'). These were simple changes that required minimal effort.
2. The reconcile step, the final step in our approach, merges similar events that may have been missed by earlier anonymization rules. If there are multiple execution events that differ by one word in the same position and there are at least five of these events, then these execution events are merged. This heuristic performs relatively well on logs from the above four applications. However, this heuristic might need to be adjusted for other types of applications.

5.3. Studying the characteristics of log files

To gain a better understanding of the performance of our approach and the other log abstraction approach, we examine the properties of the studied log files. We want to determine the effect of the complexity of a log file on the performance of a log abstraction approach. We assess the complexity of a log file by calculating the Shannon entropy metric for that file [33]. The entropy metric assesses the amount of information contained in each log file. The metric measures the uncertainty that is related to information. For example, suppose we monitored the output of an application that emitted four execution events, A, B, C, or D. As we wait for the next execution event, we are uncertain as to which event the application will produce (i.e., we are uncertain about the distribution of the output). Once we see an event outputted, our uncertainty decreases. We now have a better idea about the distribution of the output; this reduction of uncertainty has given us information. Shannon proposed to measure the amount of uncertainty using entropy. Shannon's entropy H_n is defined as

$$H_n(P) = -\sum_{k=1}^{n} (p_k \times \log_2 p_k)$$

where $p_k \geqslant 0$, $\forall k \in 1, 2, 3, \ldots, n$ (n is the number of events) and $\sum_{k=1}^{n} p_k = 1$.

DOI: 10.1002/smr

By defining the amount of uncertainty in a distribution, H_n describes the minimum number of bits required to uniquely distinguish the distribution. In other words, it defines the best possible compression for the distribution (i.e., the output of the application).

We measure the information contained in a log file by calculating Shannon's entropy on the distribution of execution events as reported by our approach. p_k is the probability that a log line will be assigned to the execution event k. To compare log files with a different number of events, we use the normalized Shannon entropy that divides the entropy value by the log of the number of events (i.e., $H_n(P)/\log_2 n$). If all log lines belong to different execution events ($p_k = 1/n$), we achieve maximum entropy. On the other hand, if all log lines belong to the same execution event i (i.e., $p_k = 0$ for $k \neq i$), we achieve minimal entropy. The larger the entropy, the more complex the log file. Table XIII tabulates the normalized entropy for the four log files (0 is minimal entropy and 1 is maximal entropy). The table shows that the Blue Gene/L and LoadSim log files are the least complex files, whereas App 1 log file is the most complex. A comparison of the entropy results shown in Table XIII and the performance of both approaches, in Table XII, indicate that the performance of an approach is independent of the complexity of a log file.

In addition to measuring the entropy of each log file, we calculated the number of log lines in each log file, the number of identified execution events, the percentage of log lines that are abstracted to the most occurring execution event (Top Event), and the percentage of log lines that are abstracted to the top 10 most occurring execution events (see Table XIV). The metrics shown in the table help explain the entropy values shown earlier in Table XIII. In particular, the higher the top 10 column in Table XIV, the lower the entropy due to the limited variability in the occurrence of execution events. For example, for the LoadSim log file, which has the lowest entropy, the top 10 column indicates that 86% of all log lines are abstracted to just 10 execution events. A closer analysis of the log file shows that LoadSim outputs a particular log line '$v: User is keeping a total of $v messages open (Max = $v)' each time the system status changes.

We also note that even though the Blue Gene/L log file contains more execution events, it is less complex than the App 1 logs. A manual analysis of the log files for both application indicated that the differences are due to two different logging styles: Blue Gene/L logs record the faults from different types of sources (e.g., disks, networks, printers, etc.). When a fault happens (e.g., disk

Table XIII. Shannon's entropy for the log files of the studied applications.

Application	App 1	App 2	LoadSim	Blue Gene/L
Entropy	0.71	0.52	0.39	0.40

Table XIV. Detailed analysis of the content of the log lines for the studied applications.

System	Log lines	Execution events	Top event (%)	Top 10 events (%)
App 1	723 608	396	5.85	41.1
App 2	1 668 876	338	12.8	70.5
LoadSim	67 651	163	60.5	86.3
Blue Gene/L	2 994 986	2918	35.9	74.3

DOI: 10.1002/smr

full), the fault will persist for a period of time until either the fault is resolved or the process is aborted. Therefore, we expect to see hundreds of adjacent log lines (e.g., disk full) that report the same message within a time period. The App 1 logs record the execution of different scenarios of a complex application; therefore, we expect a higher variability in the execution events.

6. CONCLUSION

Much of the research in dynamic analysis focuses on instrumenting an application and producing tracing logs. All too often instrumenting an application is not possible in a production setting due to performance overhead, lack of system knowledge, and the inaccessibility of the source code. On the other hand, execution logs are commonly available for most enterprise applications and could be used to study the dynamic behavior of complex applications.

It is difficult to analyze the log files produced by enterprise applications. Such logs are very large with millions of lines in them. The logs rarely follow standard logging formats. Techniques are needed to preprocess log files and identify the internal structure of each log line. Abstraction log lines abstracted to execution events helps to comprehend and summarize log files

In this paper, we developed an approach that recognizes the internal structure of log lines. The approach uses clone detection techniques to abstract each log line to its corresponding execution event. We conducted a case study using logs from four enterprise applications that are developed by three different organizations. Our case study shows that our approach performs well on large log files and has very high precision and recall.

ACKNOWLEDGEMENTS

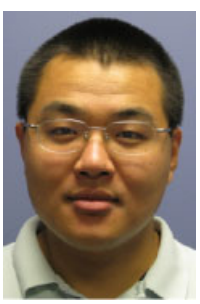
We are grateful to Research In Motion (RIM) for providing access to the execution logs of two of the enterprise applications used in our case study. The findings and opinions in this paper belong solely to the authors and are not necessarily those of RIM. Moreover, our results do not in any way reflect the quality of RIM's products. We thank Jon Stearley from Sandia National Laboratories for providing us the Blue Gene/L logs. We acknowledge the insightful and detailed comments offered by the anonymous reviewers.

REFERENCES

1. Summary of Sarbanes-Oxley Act of 2002. http://www.soxlaw.com/ [22 June 2008].
2. Apache Custom Log Format. http://www.loganalyzer.net/log-analyzer/apache-custom-log.html [22 June 2008].
3. Microsoft IIS W3C Extended Log File Format. http://www.microsoft.com/technet/prodtechnol/WindowsServer2003/Library/IIS/676400bc-8969-4aa7-851a-9319490a9bbb.mspx?mfr=true [22 June 2008].
4. SNMP Logs. http://www.ietf.org/rfc/rfc1157.txt [22 June 2008].
5. Dale R, Somers HL, Moisl H (eds.). *Handbook of Natural Language Processing*. Marcel Dekker: New York NY, U.S.A., 2000.
6. de la Higuera C. A bibliographical study of grammatical inference. *Pattern Recognition* 2005; **38**(9):1332–1348.
7. Damásio CV, Fröhlich P, Nejdl W, Pereira LM, Schroeder M. Using extended logic programming for alarm-correlation in cellular phone networks. *Applied Intelligence* 2002; **17**(2):187–202.
8. Hansen SE, Atkins ET. Automated system monitoring and notification with swatch. *LISA '93*: *Proceedings of the Seventh USENIX Conference on System Administration*, USENIX Association, 1993; 145–152.
9. Ley W, Ellerman U. http://www.cert.dfn.de/eng/logsurf/, 1995.
10. Vaarandi R. Simple event correlator for real-time security log monitoring. *Hakin9 Magazine* 2006; **6**(1):28–39.
11. Gupta M, Subramanian M. Preprocessor algorithm for network management codebook. *ID'99*: *Proceedings of the First Conference on Workshop on Intrusion Detection and Network Monitoring*, USENIX Association, 1999; 93–102.

12. Klinger S, Yemini S, Yemini Y, Ohsie D, Stolfo S. A coding approach to event correlation. *Proceedings of the Fourth International Symposium on Integrated Network Management IV*. Chapman & Hall: London U.K., 1995; 266–277.
13. Yemini SA, Sliger S, Ohsie D. High speed and robust event correlation. *IEEE Communications Magazine* 1996; 82–90.
14. Gurer DW, Khan I, Ogier R. *An Artificial Intelligence Approach to Network Fault Management*. SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, U.S.A., 1996.
15. Huard J. Probabilistic reasoning for fault management on xunet. *Technical Report*, AT&T Bell Labs, 1994.
16. Huard J, Lazar A. Fault isolation based on decision-theoretic troubleshooting. *Technical Report*, Center for Telecommunications Research, Columbia University, 1996.
17. Lin A. A hybrid approach to fault diagnosis in network and system management. *HP Technical Report*, 1998.
18. Sheppard JW, Simpson WR. Improving the accuracy of diagnostics provided by fault dictionaries. *VTS '96*: *Proceedings 14th IEEE VLSI Test Symposium* (*VTS '96*). IEEE Computer Society Press: Silver Spring MD, 1996; 180–185.
19. Stearley J. Towards informatic analysis of syslogs. *CLUSTER '04*: *Proceedings 2004 IEEE International Conference on Cluster Computing*. IEEE Computer Society: Silver Spring MD, 2004; 309–318.
20. Vaarandi R. A data clustering algorithm for mining patterns from event logs. *Proceedings 2003 IEEE Workshop on IP Operations and Management*, 2003; 119–126.
21. Wietgrefe H, Tochs K. Using neural networks for alarm correlation in cellular phone networks. *International Workshop on Applications of Neural Networks in Telecommunications*, 1997; 248–255.
22. Cordy JR. Comprehending reality: Practical challenges to software maintenance automation. *Proceedings IEEE 11th International Workshop on Program Comprehension*, *IWPC 2003* (Keynote Paper), May 2003; 196–206.
23. Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. *Proceedings of the International Conference on Software Maintenance*, 1998; 368–377.
24. Kontogiannis K. Evaluation experiments on the detection of programming patterns using software metrics. *WCRE '97*: *Proceedings of the Fourth Working Conference on Reverse Engineering* (*WCRE '97*). IEEE Computer Society Press: Silver Spring MD, 1997; 44–54.
25. Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. *ICSM '99*: *Proceedings IEEE International Conference on Software Maintenance*. IEEE Computer Society Press: Silver Spring MD, 1999; 109–118.
26. Kamiya T, Kusumoto S, Inoue K. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 2002; **28**(7):654–670.
27. Jiang ZM, Hassan AE. A framework for studying clones in large software systems. *Proceedings Seventh IEEE International Working Conference on Source Code Analysis and Manipulation* (*SCAM 2007*), 2007; 203–212.
28. Kapser C, Godfrey MW. Improved tool support for the investigation of duplication in software. *ICSM '05*: *Proceedings 21st IEEE International Conference on Software Maintenance* (*ICSM'05*). IEEE Computer Society Press: Silver Spring MD, 2005; 305–314.
29. Livieri S, Higo Y, Matsushita M, Inoue K. Analysis of the Linux kernel evolution using code clone coverage. *MSR '07*: *Proceedings of the Fourth International Workshop on Mining Software Repositories*, 2007.
30. Supercomputer Event Logs. http://www.cs.sandia.gov/~jrstear/.logs-alpha1/ [22 June 2008].
31. Oliner A, Stearley J. What supercomputers say: A study of five system logs. *DSN '07*: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society Press: Silver Spring MD, 2007; 575–584.
32. Teiresias. http://cbcsrv.watson.ibm.com/Tspd.html.
33. Weaver S. *The Mathematical Theory of Communication.* University of Illinois Press: Urbana, 1949.

AUTHORS' BIOGRAPHIES



Zhen Ming Jiang is a PhD student at Queen's University in Canada. He received both his BMath and MMath degrees from the University of Waterloo in Canada. His research interests include dynamic analysis, software performance engineering, source code duplication, mining software repositories (MSR) and reverse engineering.

DOI: 10.1002/smr

Ahmed E. Hassan is an Assistant Professor with the School of Computing at Queen's University in Canada. His research interests include Mining Software Repositories (MSR) and Performance Engineering. He spent the early part of his career helping architect the Blackberry wireless platform at Research In Motion (RIM). He contributed to the development of protocols, simulation tools, and software to ensure the scalability and reliability of RIM's global infrastructure. He previously worked for IBM Research at the Almaden Research Lab in San Jose and the Computer Research Lab at Nortel Networks (BNR) in Ottawa. He spearheaded the organization and creation of the International Working Conference on Mining Software Repositories (MSR) and its associated research community. He recently co-edited a special issue of the IEEE Transaction on Software Engineering (TSE) on MSR. (http://www.cs.queensu.ca/∼ahmed)

Gilbert E. Hamann is a Senior Performance Analyst at Research In Motion (RIM). He has a special interest in the real-world scalability of system architectures, designs, and frameworks. This interest has been applied in understanding and improving communication systems, healthcare systems and logistics systems that are globally distributed. In his early career he enjoyed pioneering software techniques and tools that provided computer access for people with disabilities.

Parminder Flora is currently the Manager of the Performance Engineering Team for the BlackBerry Enterprise Server at Research In Motion (RIM). He founded the team in 2001 and has overseen its growth to over 20 people. He holds a Bachelor's of Computer Engineering from McMaster University and has been involved for over 10 years in performance engineering within the telecommunication field. His passion is to ensure that Research In Motion (RIM) provides enterprise class software that exceeds customer expectations for performance and scalability.

DOI: 10.1002/smr