A Lightweight Algorithm for Message Type Extraction in System Application Logs

Adetokunbo Makanju, *Member*, *IEEE*, A. Nur Zincir-Heywood, *Member*, *IEEE*, and Evangelos E. Milios, *Senior Member*, *IEEE*

Abstract—Message type or message cluster extraction is an important task in the analysis of system logs in computer networks. Defining these message types automatically facilitates the automatic analysis of system logs. When the message types that exist in a log file are represented explicitly, they can form the basis for carrying out other automatic application log analysis tasks. In this paper, we introduce a novel algorithm for carrying out message type extraction from event log files. IPLoM, which stands for *Iterative Partitioning Log Mining*, works through a 4-step process. The first three steps hierarchically partition the event log into groups of event log messages or event clusters. In its fourth and final stage, IPLoM produces a message type description or line format for each of the message clusters. IPLoM is able to find clusters in data irrespective of the frequency of its instances in the data, it scales gracefully in the case of long message type patterns and produces message type descriptions at a level of abstraction, which is preferred by a human observer. Evaluations show that IPLoM outperforms similar algorithms statistically significantly.

Index Terms—Algorithms, experimentation, event log mining, fault management, clustering

1 Introduction

The goal of autonomic computing as espoused by IBM's senior vice president of research, Paul Horn in March 2001 can be defined as the goal of building self-managing computing systems [1]. The four key concepts of self-management in autonomic computing are self-configuration, self-optimization, self-healing, and self-protection. Given the increasing complexity of computing infrastructure which is stretching to its limits the human capability to manage it, the goal of autonomic computing is a desirable one. However, it is a long-term goal, which must first start with the building of computing systems, which can automatically gather and analyze information about their states to support decisions made by human administrators [1].

Event logs generated by applications that run on a system consist of independent lines of text data, which contain information that pertains to events that occur within a system. This makes them an important source of information to system administrators in fault management and for intrusion detection and prevention. With regard to autonomic systems, these two tasks are important cornerstones for self-healing and self-protection, respectively. Therefore, as we move toward the goal of building systems that are capable of self-healing and self-protection, an important step would be to build systems that are capable of automatically analyzing the contents of their log files, in addition to measured system metrics [2], [3], to provide useful information to the system administrators.

• *The authors are with the Faculty of Computer Science, Dalhousie University, 6050 University Ave., Halifax, NS B3H 4R2, Canada. E-mail: {makanju, zincir, eem}@cs.dal.ca.*

Manuscript received 3 May 2010; revised 22 Oct. 2010; accepted 9 June 2011; published online 23 June 2011.
Recommended for acceptance by C. Jermaine.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-05-0262.
Digital Object Identifier no. 10.1109/TKDE.2011.138.

A basic task in automatic analysis of log files is message type extraction [4], [5], [6], [7]. Extraction of message types makes it possible to abstract the unstructured content of event logs, which constitutes a key challenge to achieving fully automatic analysis of system logs. Message type descriptions are the templates on which the individual unstructured messages in any event log are built. Message types, once found, are useful in several ways:

- **Compression.** Message types can abstract the contents of system logs. We can therefore use them to obtain more concise and compact representations of log entries. This leads to memory and space savings.
- **Indexing.** Each unique message type can be assigned an *Identifier Index (ID)*, which in turn can be used to index historical system logs leading to faster searches. In [8], the authors demonstrated how message types can be used for log size reduction and indexing of the contents of event logs.
- **Model building.** The building of computational models on the log data, which usually requires the input of structured data, can be facilitated by the initial extraction of message type information. Message types are used to impose structure on the unstructured messages in the log data before they are used as input into the model building algorithm. In [9], [10], the authors demonstrate how message types can be used to extract measured metrics used for building computational models from event logs. The authors were able to use their computed models to detect faults and execution anomalies using the contents of system logs.
- **Visualization.** Visualization is an important component of the analysis of large data sets. Visualization of the contents of systems logs can be made more meaningful to a human observer by using message types as a feature of the visualization. For the visualization to be meaningful to a human

```
2005-06-03-15.42.50.823719 R02-M1-N0-C:J12-U11 RAS KERNEL INFO instruction cache parity error corrected
2005-06-03-15.42.50.982731 R02-M1-N0-C:J12-U11 RAS KERNEL INFO instruction cache parity error corrected
2005-06-06-22.41.37.357738 R20-M0-NA-C:J15-U11 RAS KERNEL INFO generating core.3740
2005-06-06-22.41.37.392258 R20-M0-NA-C:J17-U11 RAS KERNEL INFO generating core.3612
2005-06-11-19.20.25.104537 R30-M0-N9-C:J16-U01 RAS KERNEL FATAL data TLB error interrupt
2005-06-11-19.20.25.393590 R30-M0-N9-C:J16-U01 RAS KERNEL FATAL data TLB error interrupt
2005-07-01-17.52.23.557949 R22-M0-NA-C:J05-U01 RAS KERNEL INFO 458720 double-hummer alignment exceptions
2005-07-01-17.52.23.584839 R22-M0-NA-C:J03-U01 RAS KERNEL INFO 458720 double-hummer alignment exceptions
```

Fig. 1. An example system log file. Each line represents an event.

observer, the message types must be interpretable. This fact provides a strong incentive for the production of message types that have meaning to a human observer.

To give an example of what message types are, consider this line of code:

```
sprintf(message, Connection from %s port %d,
ipaddress, portnumber);
```

in a *C* program could produce the following log entries:

```
"Connection from 192.168.10.6 port 25"
"Connection from 192.168.10.6 port 80"
"Connection from 192.168.10.7 port 25"
"Connection from 192.168.10.8 port 21."
```

These four log entries would form a cluster (group) or event type in the event log and can be represented by the message type description (or line format):

```
"Connection from * port *."
```

The wildcards "*" represent message variables. We will adopt this representation in the rest of our work. Determining what constitutes a message type might not always be as simple as this example might suggest. Consider the following messages produced by the same print statement. `"Link 1 is up,"` `"Link 1 is down,"` `"Link 3 is down,"` `"Link 4 is up."` The most logical message type description here is `"Link * is *,"` however from a analysis standpoint having two descriptions `"Link * is up"` and `"Link * is down"` maybe preferable. There may also be other cases where messages produced by different print statements could form single logical message types. However, for the most part, message types will usually correspond to messages produced by the same print statement, so we retain our representation for simplicity.

The goal of message type extraction is to find the representations of the message types that exist in a log file. This problem is well attested to in the literature but there is as yet no standard approach to the problem [9]. Techniques for automatically mining these line patterns from event logs have been based on the Apriori algorithm [11] for frequent item sets from data, e.g., Simple Log File Clustering Tool (SLCT) [12] and Loghound [13], or other line pattern discovery techniques like Teiresias [14] designed for other domains [7]. SLCT, Loghound, and Teiresias as algorithms are aimed toward the discovery of frequent textual patterns.

In this paper, we introduce *Iterative Partitioning Log Mining* (IPLoM), a novel algorithm for the mining of event type patterns from event logs. Unlike previous algorithms, IPLoM is not primed toward the finding of only frequent textual patterns, but instead IPLoM's aim is to find all possible patterns. IPLoM works through a 3-step partitioning process, which partitions a log file into its respective clusters. In a fourth and final stage, the algorithm produces a cluster description for each leaf partition of the log file. These cluster descriptions then become event type patterns discovered by the algorithm. IPLoM is able to find clusters in the data irrespective of the frequency of its instances and it scales gracefully in face of long message type patterns and it produces message type descriptions at a level of abstraction, which is preferred by a human observer. In our experiments, we compared the outputs of IPLoM, SLCT, Loghound, and Teiresias on seven different event log files, making up over 1 million log events, against message types produced manually on the event log files by our Faculty's tech support group. Results demonstrate that IPLoM consistently outperforms the other algorithms. It was able, in the best case, to produce approximately 70 percent of the manually produced message types compared to 36 percent for the best existing algorithm.

The rest of this paper is organized as follows: Section 2 discusses previous work in event type pattern mining and categorization. Section 3 outlines the proposed algorithm and the methodology to evaluate its performance. Section 4 describes the results whereas Section 5 presents the conclusion and the future work.

2 Background and Previous Work

We begin this section by first defining some of the terminology used in this paper. We then discuss previous related work in the area of event log clustering and message type extraction.

2.1 Definitions

- **Event log.** A text-based audit trail of events that occur within the system or application processes on a computer system (Fig. 1).
- **Event.** An independent line of text within an event log which details a single occurrence on the system, (Fig. 2). An event typically contains not only a *message* but other fields of information like a *Date*, *Source*, and *Tag* as defined in the `syslog` Request for Comment (RFC) [15]. For message type extraction, we are only interested in the *message* field of the event. This is why events are sometimes referred to in the literature as messages. In Fig. 2, the first five fields (delimited by whitespace) represent the *Timestamp*, *Host*, *Class*,

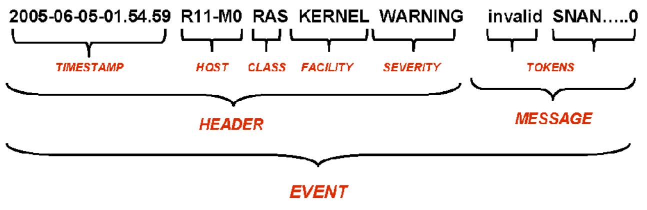


Fig. 2. An example system log event.

Facility, and *Severity* of each event. We omit these types of fields from the message type extraction process as they are already sufficiently structured. However, they are still useful for further log analysis, e.g., the *Timestamp* and *Host* fields for time series analysis of the unique message types extracted.

- **Token.** A single word delimited by white space within the *message* field of an event. For example, in Fig. 2, the words *invalid* and $SNA \ldots 0$ are tokens in that message.
- **Event size.** The number of individual tokens in the "message" field of an event. The event in Fig. 2 has an event size of 2.
- **Event cluster/message type.** These are *message* fields of entries within an event log produced by the same print statement. Nonoverlapping consecutive pairs of lines in the log example shown in Fig. 1 belong to the same event cluster. Due to the subjectivity of determining what constitutes a message type, it is possible that a human observer might consider messages produced by a single message type as belonging to different message types or treat messages produced by different print statements as belonging to the same message type. It is also possible that the same print statement is present in different parts of the code, producing different messages types with the same message type description. However, we consider these scenarios as relatively rare, so we will use this definition for the sake of simplicity.
- **Cluster description/message type description/line format.** A textual template containing wildcards which represents all members of an event cluster. The messages in the third and fourth lines of Fig. 1 have a cluster description of "*generating* $*$."
- **Constant token.** A token within the *message* field of an event which is not represented by a wildcard value in its associated message type description. The token *generating* in the third line of Fig. 1 is a constant token.
- **Variable token.** A token within the *message* field of an event which is represented by a wildcard value in its associated message type description. The token *core.3740* in the third line of Fig. 1 is a variable token.

2.2 Previous Work

Data clustering as a technique in data mining or machine learning is a process whereby entities are sorted into groups called clusters, where members of each cluster are similar to each other and dissimilar from members of other groups. Clustering can be useful in the interpretation and classification of data sets too large to analyze manually. Clustering therefore can be a useful first step in the automatic analysis of event logs.

If each textual line in an event log is considered a data point and its individual words considered attributes, then the clustering task reduces to one in which similar log messages are grouped together. For example, the log entry *Command has completed successfully* can be considered a 4-dimensional data point with the following attributes *"Command," "has," "completed," "successfully."* However, as stated in [12], traditional clustering algorithms are not suitable for event logs for the following reasons:

1. The event lines do not have a fixed number of attributes.
2. The data point attributes, i.e., the individual words or tokens on each line, are categorical. Most conventional clustering algorithms are designed for numerical attributes.
3. Traditional clustering algorithms also tend to ignore the order of attributes. In event logs, the attribute order is important.

While several algorithms like CLIQUE [16], CURE [17], and MAFIA [18] have been designed for clustering high-dimensional data, these algorithms are still not quite suitable for log files because an algorithm suitable for clustering event logs needs to not just be able to deal with high-dimensional data, but it also needs to be able to deal with data with different attribute types [12], [19].

For these reasons, several algorithms and techniques for automatic clustering and/or categorization of log files have been developed. Moreover, some researchers have also attempted to use techniques designed for pattern discovery in other types of textual data to the task of clustering event logs [7], while other researchers have determined the message types in system logs by extracting them directly from source code [9]. This approach, however, assumes access to source code and an appropriate language parser.

In [20], the authors attempt to classify raw event logs into a set of categories based on the IBM Common Base Event (CBE) format [21] using Hidden Markov Models (HMM) and a modified Naive Bayesian Model. They report 85 and 82 percent classification accuracy, respectively. While similar, the automatic categorization done in [20] is not the same as discovering event log clusters or formats. This is because the work done in [20] is a supervised classification problem, with predefined categories, while the problem we tackle is unsupervised, with the final categories not known a priori.

On the other hand SLCT [12] and Loghound [13] are two algorithms, which were designed specifically for automatically clustering log files, and discovering event formats. This is similar to our objective in this paper. Because both SLCT and Loghound are similar to the Apriori algorithm [11], they require the user to provide a support threshold value as input.

SLCT works through a three step process:

1. It firsts identifies the frequent words (words that occur more frequently than a support threshold value) or 1-item sets from the data.
2. It then extracts the combinations of these 1-item sets that occur in each line in the data set. These 1-item set combinations are cluster candidates.
3. Finally, those cluster candidates that occur more frequently than the support value are then selected as the clusters in the data set.

Loghound on the other hand discovers frequent patterns from event logs by utilizing a frequent item set mining algorithm, which mirrors the Apriori algorithm more closely than SLCT because it works by finding item sets

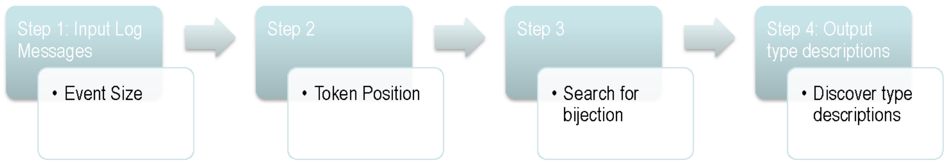

Step 1: Input Log Messages
• Event Size
Step 2
• Token Position
Step 3
• Search for bijection
Step 4: Output type descriptions
• Discover type descriptions

Fig. 3. Overview of IPLoM's processing steps.

which may contain more than *1* word up to a maximum value provided by the user. With both SLCT and Loghound, lines that do not match any of the frequent patterns discovered are classified as outliers.

SLCT and Loghound have received considerable attention and have been used in the implementation of the Sisyphus Log Data Mining toolkit [22], as part of the LogView log visualization tool [23] and in online failure prediction [24].

A comparison of SLCT against a bioinformatics pattern discovery algorithm developed by IBM called Teiresias [14] is carried out in [7]. Teiresias was designed to discover all patterns of at least a given specificity and support in categorical data.

In our work, we introduce IPLoM, a novel log-clustering algorithm. IPLoM works differently from the other clustering algorithms described above as it is not based on the Apriori algorithm and does not explicitly try to find line formats. The algorithm works by creating a hierarchical partitioning of the log data. The leaf nodes of this hierarchical partitioning of the data are considered clusters of the log data and they are used to find the cluster descriptions or line formats that define each cluster. Our experiments demonstrate that IPLoM outperforms SLCT, Loghound, and Teiresias when they are evaluated on the same data sets.

Other researchers, however, deal with lack of structure in event logs not through the use of clustering but by attempting to store log information using the same reporting standard. In [25], the authors introduce the Generic Adapter Logging Toolkit for autonomic computing. This toolkit is capable of converting messages and events from heterogeneous components with disparate logging standards and formats into a common format based on the IBM Common Base Event format [21]. The unified reporting format eases the automatic analysis of the log data in autonomic computing. The toolkit consists of three components: the runtime translator, the rule repository, and the rule builder. The CBE is, however, an IBM proprietary format.

3 Methodology

In this section, we first give a detailed description of our proposed algorithm and our methodology for testing its performance against those of previous algorithms.

3.1 The IPLoM Algorithm

The IPLoM algorithm is designed as a log data clustering algorithm. It works by iteratively partitioning a set of log messages used as training exemplars. At each step of the partitioning process, the resultant partitions come closer to containing only log messages which are produced by the same line format. At the end of the partitioning process, the algorithm attempts to discover the line formats that produced the lines in each partition. These discovered partitions and line formats are the output of the algorithm.

Our main assumptions on the kind of event logs that IPLoM is suited for are the following:

1. The events in the log contain at least one field that is an unstructured natural language description of the event. These descriptions, which we call "messages," illustrated in Fig. 2, would naturally be produced by a set of "print" statements in a program source code.
2. The exact structure of these "messages" is unknown or not well documented.

Our work is therefore relevant to any log file where these assumptions are true, not just log files that meet the format of Fig. 2. Some application event logs are well structured and well documented, e.g., Webshpere. In such cases, message type extraction may not be necessary. For example, in [26], the authors provide a use case of process mining in web services using Websphere, while in [27], the authors propose a tool for visualizing web services behavior by mining the contents of event logs. Process mining refers to the analysis of event logs as a way of monitoring adherence to business process rules. The analysis can be carried out without message type extraction due to the structured nature of the Webshpere logs, which utilize the Common Base Event model [21] for event representation.

An outline of the four steps of IPLoM is given in Fig. 3. The algorithm is designed to discover all possible line formats in the initial set of log messages and does not require a support threshold like SLCT or Loghound. As it may be sometimes required to find only line formats that have a support that exceeds a certain threshold, a file prune function (Algorithm 1) is incorporated into the algorithm. By removing the partitions that fall below the threshold value at the end of each partitioning step, we are able to produce only line formats that meet the desired support threshold at the end of the algorithm. The use of the file prune function is however optional. The following sections describe each step of the algorithm in more detail.

Algorithm 1. File_Prune Function: Prunes the partitions produced using the file support threshold.
Input: Collection $C[]$ of log file partitions.
Real number FS as file support threshold. {Range for FS is assumed to be between $0 - 1$.}
Output: Collection $C[]$ of log file partitions with support greater than FS.

```
1: for every $partition\ in\ C$ do
2:   $Supp = \frac{\#LinesInPartition}{\#LinesInCollection}$
3:   if $Supp <\ FS$ then
4:     Delete partition from $C[]$
5:   end if
6: end for
7: Return(C)
```

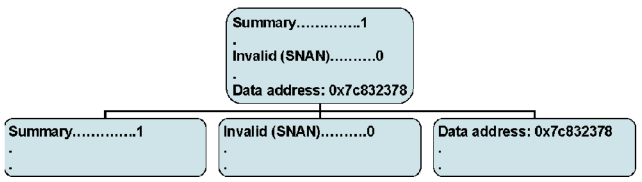


Fig. 4. IPLoM Step 1: partition by event size. Separates the messages into partitions based on the their event size.

3.2 Step 1: Partition by Event Size

The first step of the partitioning process works on the assumption that log messages that have the same message type description are likely to have the same event size. For this reason, IPLoM's first step (Fig. 4) uses the event size heuristic to partition the log messages. By partition, we mean nonoverlapping groupings of the messages. Additional heuristic criteria are used in the remaining steps to further divide the initial partitions. The partitioning process induces a hierarchy of maximum depth 4 on the messages and the number of nodes on each level is data dependent. Consider the cluster description *"Connection from* $*$*,"* which contains three tokens. It can be intuitively concluded that all the instances of this cluster, e.g., *"Connection from 255.255.255.255"* and *"Connection from 0.0.0.0"* would also contain the same number of tokens. By partitioning our data first by event size, we are taking advantage of the property of most cluster instances of having the same event size. Therefore, the resultant partitions of this heuristic are likely to contain the instances of the different clusters, which have the same event size.

Sometimes, it is possible that clusters with events of variable size exist in the event log. This scenario is explained in more detail in Section 4.7.3. Since IPLoM assumes that messages belonging to the same cluster should have the same number of tokens or event size, this step of the algorithm would separate such clusters. This does not occur too often, and variable size message types can still be found by postprocessing IPLoM's results. The process of finding variable size message types can be computationally expensive. Nevertheless, performing this process on the templates produced by IPLoM rather than on the complete log would require less computation.

3.3 Step 2: Partition by Token Position

At this point, each partition of the log data contains log messages, which are of the same size and can therefore be viewed as n-tuples, with n being the event size of the log messages in the partition. This step of the algorithm works on the assumption that the column with the least number of variables (unique words) is likely to contain words, which are constant in that position of the message type descriptions that produced them. Our heuristic is therefore to find the token position with the least number of unique values and further split each partition using the unique values in this token position, i.e., each resultant partition will contain only one of those unique values in the token position discovered, as can be seen in the example outlined in Fig. 5. A pseudocode description of this step of the partitioning process is given in Algorithm 2.

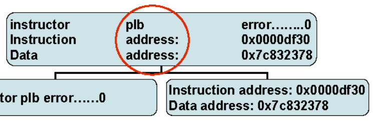


Fig. 5. IPLoM Step 2: partition by token position. Selects the token position with the least number of unique values, token position 2 in this example. Then, it separates the messages into partitions-based unique token values, i.e., "plb" and "address:", in the token position.

Algorithm 2. IPLoM Step 2: Selects the token position with the lowest cardinality and then separates the lines in the partition based on the unique values in the token position. Backtracks on partitions with lines that fall below the partition support threshold.

Input: Collection of log file partitions from Step-1.
Real number PST as partition support threshold. {Range for PST is assumed to be between $0-1$.}

Output: Collection of log file partitions derived at Step-2 C_In.

```
1:  for every log file $partition$ do {Assume lines in each
    partition have same event size.}
2:      Determine token position $P$ with lowest cardinality
        with respect to set of unique tokens.
3:      Create a partition for each token value in the set of
        unique tokens that appear in position $P$.
4:      Separate contents of partition based on unique token
        values in token position $P$. into separate partitions.
5:  end for
6:  for each partition derived at Step-2 do {}
7:      if $PSR < PS$ then
8:          Add lines from partition to Outlier partition
9:      end if
10: end for
11: File_Prune() {Input is the collection of newly created
    partitions}
12: Return() {Output is collection of pruned new partitions}
```

The memory requirement of unique token counting is a potential concern with the algorithm. While the problem of unique token counting is not specific to IPLoM, we believe IPLoM has an advantage in this respect. Since IPLoM partitions the database, only the contents of the partition being handled need be stored in memory. This greatly reduces the memory requirements of the algorithm. Moreover, other workarounds can be implemented to further reduce the memory requirements. For example, in this Step 2 of the algorithm, by determining an upper bound (UB) on the lowest token count in Step 1, we can drastically reduce the memory requirements of this step, further counts of unique tokens in any token position that exceeds the upper bound can be eliminated. However, in this work, our aim is to make a proof of concept so we left the implementation of such code optimization techniques for future work.

Despite the fact that we use the token position with the least number of unique tokens, it is still possible that some of the values in the token position might actually be variables in the original message type descriptions. While an error of this type may have little effect on Recall, it could adversely affect Precision. To mitigate the effects of this error, a partition support ratio (PSR) for each partition produced could be introduced. The PSR is calculated using

```
Fan speeds 3552 3552 3391 4245 3515 3497
Fan speeds 3552 3534 3375 4787 3515 3479
Fan speeds 3552 3534 3375 6250 3515 3479
Fan speeds 3552 3534 3375 **** 3515 3479
Fan speeds 3311 3534 3375 4017 3515 3479
```

Fig. 6. Example messages illustrating 1-M, M-1, and M-M relationships.

(1) in regard to the original partition that it was derived from. We can then define a partition support ratio threshold (PST). We group any partition with a PSR that falls below the PST into one partition (Algorithm 2). The intuition here is that a child partition that is produced using a variable token value may not have enough lines to exceed a certain percentage (the partition support ratio threshold) of the log messages in the parent partition. It should be noted that this threshold is not necessary for the algorithm to function and is only introduced to give the system administrators the flexibility to influence the partitioning based on expert knowledge they may have and avoid errors in the partitioning process

$$PSR = \frac{\#LinesInChildPartition}{\#LinesInParentPartition}. \qquad (1)$$

3.4 Step 3: Partition by Search for Bijection

In the third and final partitioning step, we partition by searching for bijective relationships between the set of unique tokens in two token positions selected using a heuristic as described in Algorithm 3. Consider the example the messages below as a log partition.

```
Command has completed successfully
Command has been aborted
Command has been aborted
Command has been aborted
Command failed on starting.
```

This partition has event size equal to *4*. We need to select two token positions to perform the search for bijection on. The first token position has one unique token, {Command}. The second token position has two unique tokens, {has, failed}. The third token position has three unique tokens, {completed, been, on}. While the fourth token position has three unique tokens, {successfully, aborted, starting}. We notice in this example that token count *3* appears most frequently, twice, once in position *3* and once in position *4*. The heuristic would therefore select token positions *3* and *4* in this example.

Algorithm 3. IPLoM Step 3: Selects the two token positions and then separates the lines in the partition based on the relational mappings of unique values in the token positions. Backtracks on partitions with lines that fall below the partition support threshold.
Input: Collection of partitions from Step 2. {Partitions of event size 1 or 2 are not processed here}
Real number CT as cluster goodness threshold. {Range for CT is assumed to be between $0-1$.}
Output: Collection of partitions derived at Step-3.

```
1:  for every log file $partition$ do
2:    if $CGR >= CT$ then {See (2)}
3:      Add partition to collection of output partitions
4:      Move to next partition.
5:    end if
6:    Determine token positions using heuristic as $P1$ and
      $P2$. {Heuristic is explained in the text. We assume
      token position $P1$ occurs before $P2$.}
7:    Determine mappings of unique token values $P1$ in
      respect of token values in $P2$ and vice versa.
8:    if mapping is $1-1$ then
9:      Create partitions for event lines that meet each $1-1$
        relationship.
10:   else if mapping is $1-M$ or $M-1$ then
11:     Determine variable state of $M$ side of relationship.
12:     if variable state of $M$ side is $CONSTANT$ then
13:       Create partitions for event lines that meet
          relationship.
14:     else {variable state of $M$ side is $VARIABLE$}
15:       Create new partitions for unique tokens in $M$ side
          of the relationship.
16:     end if
17:   else {mapping is $M-M$}
18:     All lines with meet $M-M$ relationships are placed
        in one partition.
19:   end if
20: end for
21: for each partition derived at Step-3 do {}
22:   if $PSR <\ PS$ then
23:     Add lines from partition to Outlier partition
24:   end if
25: end for
26: File_Prune() {Input is the collection of newly created
    partitions}
27: Return() {Output is collection of pruned new partitions}
```

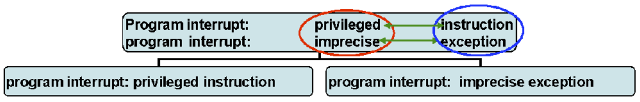

Program interrupt:
program interrupt:
privileged
imprecise
instruction
exception
program interrupt: privileged instruction
program interrupt: imprecise exception

Fig. 7. IPLoM Step-3: partition by search for bijection.

To summarize the steps of the heuristic, we first determine the number of unique tokens in each token position of a partition. We then determine the most frequently occurring token count among all the token positions. This value must be greater than 1. The token count that occurs most frequently is likely indicative of the number of message types that exist in the partition. If this is true, then a bijective relationship should exist between the tokens in the token positions that have this token count. Once the most frequently occurring token count value is determined, the token positions chosen will be the first two token positions, which have a token count value equivalent to the most frequently occurring token count.

A bijective function is a *1-1* relation that is both injective and surjective. When a bijection exists between two elements in the sets of tokens, this usually implies that a strong relationship exists between them and log messages that have these token values in the corresponding token positions are separated into a new partition.

Sometimes, the relations found are not *1-1* but *1-M*, *M-1*, and *M-M*. In the example given in Fig. 7, the tokens *privileged* and *instruction* with the tokens *imprecise* and *exception* have a 1-1 relationship because all lines that contain the tokens

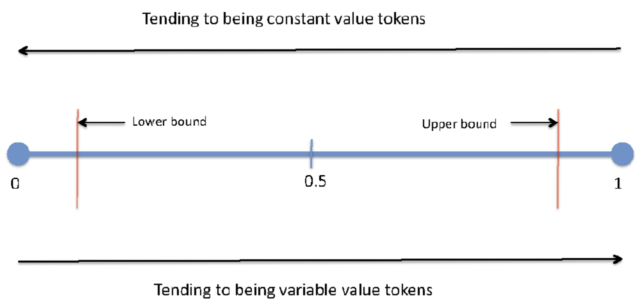

Tending to being constant value tokens
Lower bound
Upper bound
0
0.5
1
Tending to being variable value tokens

Fig. 8. Deciding on how to treat 1-M and M-1 relationships. This procedure is implemented in the Get_Rank_Position function.

privileged and *imprecise* in position 2 also, respectively, contain the tokens *instruction* and *exception* in position 3 and vice versa. Consider the event messages given in Fig. 6 below to illustrate 1-M, M-1, and M-M relationships. If token positions 2 and 3 are chosen by the heuristic, we would have a 1-M relationship with tokens *speeds*, *3,552* and *3,311* as all lines that contain the token *speeds* in position 2 have either tokens *3,552* or *3,311* in position 3, a M-1 relationship will be the reverse of this scenario. On the other hand, if token positions 3 and 4 are chosen by the heuristic, we would have a M-M relationship.

It is obvious that no discernible relationship can be found with the tokens in the chosen positions. Token *3,552 (in position 3)* maps to tokens *3,552 (in position 4)* and *3,534*. On the other hand, token *3,311* also maps to token *3,534*, this makes it impossible to split these messages using their token relationships. It is a scenario like this that we refer to as a M-M relationship.

In the case of *1-M* and *M-1* relations, the M side of the relation could represent variable values (so we are dealing with only one message type description) or constant values (so each value actually represents a different message type description). The diagram in Fig. 8 describes the simple heuristic that we developed to deal with this problem. Using the ratio between the number of unique values in the set and the number of lines that have these values in the corresponding token position in the partition, and two threshold values, a decision is made on whether to treat the M side as consisting of constant values or variable values. *M-M* relationships are iteratively split into separate *1-M* relationships or ignored depending on if the partition is coming from Steps 1 or 2 of the partitioning process, respectively.

Before partitions are passed through the partitioning process of Step 3 of the algorithm, they are evaluated to determine if they already form good clusters. To do this, a cluster goodness ratio threshold (CGT) is introduced into the algorithm. The cluster goodness ratio (CGR) is the ratio of the number of token positions that have only one unique value to the event size of the lines in the partition, according to (2). In the example in Fig. 7, the partition to be split has four token positions. Of these four, the first and second have only one unique value, i.e., "Program" and "Interrupt," respectively. Therefore, the CGR for this partition will be $\frac{2}{4}$. Partitions that have a value higher than the CGT are considered good clusters and are not partitioned any further in this step. Just as in Step 2 the PSR can be used to backtrack on the partitioning at the end of Step 3. While the backtracking is optional as with Step 2, we do advice that it is carried out to deal with errors when they occur

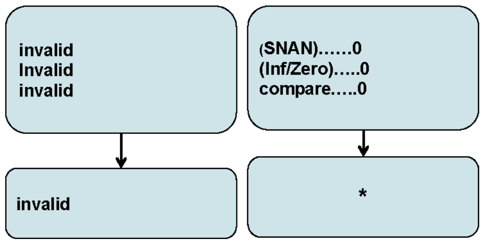


Fig. 9. IPLoM Step 4: discover message type descriptions. If the cardinality of the unique token values in a token position is equal to 1, then that token position is represented by that token value in the template. Else, we represent the token position with an "*".

$$CGR = \frac{\#TokenPositionsWithOneUniqueTokenInPartition}{\#EventSizeOfPartition}. \tag{2}$$

3.5 Step 4: Discover Message Type Descriptions (Line Formats) from Each Partition

In this step of the algorithm, partitioning is complete and we assume that each partition represents a cluster, i.e., every log message in the partition was produced using the same line format. A message type description or line format consists of a line of text where constant values are represented literally and variable values are represented using wildcards. This is done by counting the number of unique tokens in each token position of a partition. If a token position has only one value then it is considered a constant value in the line format, while if it is more than one then it is considered a variable. This process is illustrated in Fig. 9.

Since our goal is to find all message types that may exist in an event log or ensure that the presence of every message type contained in an event log is reflected in the message types produced, we are not concerned about the occurrence of *"outliers"* interfering with the formats produced at this step. Hence, we set the threshold for determining a variable token position as any token position with more than one unique token.

3.6 Algorithm Parameters

In this section, we give a brief overview of the parameters/thresholds used by IPLoM. The fact that IPLoM has several parameters, which can be used to tune its performance, provides flexibility for the system administrators since this gives them the option of using their expert knowledge when they see it necessary

- **File support threshold (FST).** Ranges between [0,1]. It reduces the number of clusters produced by IPLoM. Any cluster whose instances have a support value less than this threshold is discarded. The higher this value is set to, the fewer the number of clusters that will be produced. This parameter is similar to the *support threshold* defined for SLCT and Loghound.
- **Partition support threshold.** Ranges between [0,1]. It is essentially a threshold that controls backtracking. Based on our experiments, the guideline is to set this parameter to very low values, i.e., <0.05, for optimum performance.
- **Upper_bound and Lower_bound.** Ranges between [0,1]. They control the decision on how to treat the

M side of relationships in Step 2. Lower_Bound should usually take values <0.5 while Upper_Bound takes values >0.5.
- **Cluster goodness threshold.** Ranges between [0,1]. It is used to avoid further partitioning. Its optimal setting should lie in the range of 0.3-0.6.

4 Results

Our goal in the design of IPLoM was threefold. The first was to design an algorithm that is able to find all message types that may exist in a given log file. The second was to give every message type an equal chance of being found irrespective of the frequency of its instances in the data. Our third was to design an algorithm that will produce message types at an abstraction level preferred by a human observer. We therefore begin our discussion in this section by first describing the setup of our experiments in Section 4.1 and then providing results that show how these goals have been met using a default scenario for running the algorithm, i.e., when we want to find all message types in Section 4.2. We also provide results on resource consumption (CPU and Memory) for the SLCT, Loghound and IPLoM in Section 4.2. In Sections 4.3 and 4.4, we show how varying the line support threshold (FST) using low absolute counts and percentage values, respectively, affects the results of the algorithms. SLCT, Loghound, and Teiresias need a line support threshold to produce clusters, while IPLoM does not. For SLCT and Loghound, this support value can be specified either as a percentage of the number of events in the event log or as an absolute value. For this reason, we run two sets of experiments using support values specified as percentages and as absolute values. In either case, we set these support values low because intuitively this allows for finding most of the clusters in the data, which is one of our goals. In Section 4.5, we present results of parameter sensitivity analysis. In Section 4.6, we present a case study testing IPLoM on the logs from one of the world's fastest supercomputers. In Section 4.7, we discuss our analysis of the performance limits of IPLoM.

4.1 Experimental Setting

All our experiments were run on an iMac7 desktop computer running Mac OS X 10.5.6. The machine has an Intel Core 2 Duo processor with a speed of 2.4 GHz and 2 GB of memory. In order to evaluate the performance of IPLoM, we selected open source implementations of algorithms previously used in system/application log data mining. For this reason, SLCT, Loghound, and Teiresias were selected. We therefore tested the four algorithms against seven log data sets, which we compiled from different sources, Table 1 gives an overview of the data sets used. The message types in Table 1 were derived manually. The HPC log file is a publicly available data set collected on high performance clusters at the Los Alamos National Laboratory in New Mexico, USA [28]. The Access, Error, System, and Rewrite data sets were collected on our faculty network at Dalhousie, while the Syslog and Windows files were collected on servers owned by a large ISP working with our research group. Due to privacy issues, we are not able to make the Dalhousie and ISP data available to the public.

The message type descriptions of these seven data sets were produced manually by Tech Support members of the Dalhousie Faculty of Computer Science. Table 1 gives the number of clusters identified in each file manually. Some form of automation could have been utilized in the labeling process, for example, regular expressions, however the decision on what constitutes a message type was completed manually. Again due to privacy issues, we are able to provide manually produced cluster descriptions only for the HPC data.[1] These cluster descriptions then became our gold standard, against which to measure the performance of the algorithms as an information retrieval (IR) task. As in classic IR, our performance metrics were Precision, Recall, and F-Measure, which are described by (3), (4), and (5), respectively. The terms TP, FP, and FN in the equations are the number of True Positives, False Positives, and False Negatives, respectively. Their values are derived by comparing the set of manually produced message type descriptions to the set of retrieved formats produced by each algorithm. In our evaluation, a message type description is still considered an FP even if it matches a manually produced message type description to some degree, the match has to be exact for it to be considered a TP.

For completeness, we evaluated our Precision, Recall, and F-Measure values using three different methods. In the two methods, we evaluated the results of the algorithms as a classification problem. Using the manually produced event types as classes, we evaluated how effectively the automatically produced classification results matched the manually produced labels. This classification evaluation produced Microaverage and Macroaverage results. These results are referred to as *"Micro"* and *"Macro"* in the results section. In the third method, the manually produced message type descriptions are compared against the automatically produced ones. This evaluation method is called "IR" in the results section. We believe that the "IR" method evaluation satisfies our goals better as it tests the goodness of the clusters produced. The "IR" method evaluates the results strictly based on how well the message types produced match the manually produced message types. The next section gives more details about the results of our experiments

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

TABLE 1
Log Data Statistics

Name	Description	# Messages	# Msg. Types
HPC	High Performance Cluster (Los Alamos)	433,490	106
Syslog	OpenBSD Syslog	3,261	60
Windows	Windows Oracle Application Log	9,664	161
Access	Apache Access Log	69,902	14
Error	Apache Error Log	626,411	166
System	OS X Syslog	24,524	9
Rewrite	Apache mod_rewrite Log	22,176	10

1. Descriptions are available for download from http://web.cs.dal.ca/~makanju/iplom.

TABLE 2
Algorithm Parameters

Algorithm	Parameter	Value
SLCT and Loghound		
	Support Threshold (-s)	0.01 - 0.1
	Seed (-i)	5
Teiresias		
	Sequence Version	On
	L (min. no. of non wild card literals in pattern)	1
	W (max. extent spanned by L consecutive non wild card literals)	15
	K (Min. no. of lines for pattern to appear in)	2
IPLoM		
	File Support Threshold (Percentage)	0 - 0.1
	File Support Threshold (Absolute)	1 - 20
	Partition Support Threshold	0
	Lower Bound	0.1
	Upper Bound	0.9
	Cluster Goodness Threshold	0.34

$$F\text{-}Measure = \frac{2 * Precision * Recall}{Precision + Recall}. \tag{5}$$

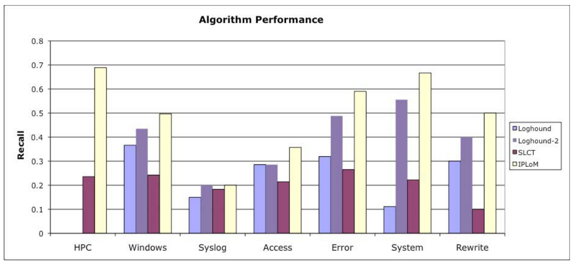
The parameter values used in running the experiments that produced our baseline results and the results in Sections 4.3 and 4.4 are provided in Table 2. The seed value for SLCT and Loghound is a seed for a random number generator used by the algorithms, all other parameter values for SLCT and Loghound are left at their default values. The parameters for Teiresias were also chosen to achieve the lowest support value allowed by the algorithm. The IPLoM parameters were all set empirically except in the case of the cluster goodness threshold and the partition support threshold.

In setting the cluster goodness threshold, we ran IPLoM on the HPC file while varying this value. The parameter was then set to the value (0.34) that gave the best result and was kept constant for the other files used in our experiments. On the other hand, the partition support threshold was set to 0 to provide a baseline performance. Such a setting for the performance threshold implies that no backtracking was done during partitioning.

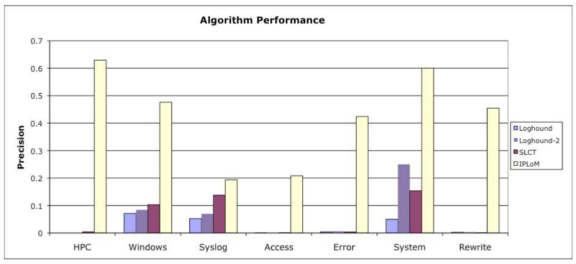
It is pertinent to note that we were unable to test the Teiresias algorithm against all our data sets. This was due to its inability to scale to the size of our data sets. This is a problem that is attested to in [7], too. Thus, in this work, Teiresias could only be tested against the Syslog data set. The memory consumption results were obtained by monitoring the processes for each algorithm by using the Unix *ps* and *grep* utilities.

4.2 Baseline Experiments

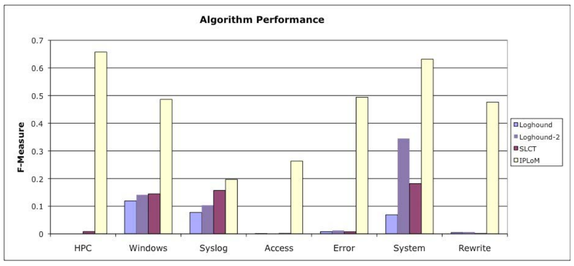
The result of our default evaluation of SLCT, Loghound, and IPLoM are shown in Fig. 10. The graphs in Fig. 10 visualize these results for Recall, Precision, and F-Measure metrics for all the algorithms using the "IR" evaluation method. Since one of our aims is to find all message types that may exist in a log file, we run this set of experiments with the lowest file support threshold possible, which is an absolute support value of *1*. SLCT and Loghound would not work efficiently with an absolute support value of *1*, so we run them with *2* instead. An absolute support value of *1* means every line/word will be considered frequent and the result of the algorithms will be reduced to the case of finding unique lines for SLCT or the case of finding all possible templates for Loghound. Both situations are not



(a) Recall



(b) Precision



(c) F-Measure

Fig. 10. Comparing algorithm IR performance at lowest support values.

TABLE 3
Log Data Event Size Statistics

Name	Min	Max	Avg.
HPC	1	95	30.7
Syslog	1	25	4.57
Windows	2	82	22.38
Access	3	13	5.0
Error	1	41	9.12
System	1	11	2.97
Rewrite	3	14	10.1

desirable. Since Teiresias worked only on the Syslog dataset, its results are not included in our analysis. Utilizing the parameter values listed in Table 2, Teiresias produced a Recall performance of 0.1, a Precision performance of 0.04, which led to an F-Measure performance of 0.06 using the IR evaluation method. By providing "IR" evaluations that compare the results of the algorithms with manually produced results, we evaluate how well we have met the third design goal, which was to design an algorithm that will produce message types at an abstraction level preferred by a human observer.

For fairness in our comparison, we provide two different evaluations for Loghound. Loghound being a frequent item set mining algorithm is unable to detect variable parts of a message type when they occur at the tail end of a message type description. For example, if we intend to find the message type description *"Error code: *,"* it is possible for Loghound to find the message type description *"Error code:"* without the trailing variable at the end. In such a situation, Loghound would not be credited with finding the message type description. We therefore provide a second set of evaluations for Loghound (referred to as Loghound-2), which adjust Loghound's results by comparing them to results when the last trailing variable in the manually produced message types are discarded.

We however state that we provide these results for information purposes only. It is our belief that considering message type descriptions where the number of trailing variables cannot be assessed is detrimental to our goal of ensuring that we find only meaningful message types at an abstraction level preferred by a human observer. When we cannot assess the number of trailing variables in message type descriptions, we lose event size, which is a means of differentiating between messages types. This leads to a loss of meaning and ambiguity. Consider these three actual examples of message types found manually *"Link *," "Link error on broadcast tree," "Link in reset."* Without the trailing variable, the first message type becomes *"Link"* which can be interpreted to mean any message that starts with the word "Link" since we do not know what the event size of the message type is. This interpretation means that we can no longer distinguish an instance of the first message type from an instance of the second or third message types. So even though we have presented Loghound-2 results where the trailing variables are not used, we believe that in practice one needs to use the trailing variables to distinguish the aforementioned differences. However, we wanted to give the benefit of doubt to Loghound for a fair comparison.

TABLE 4
Algorithm Performance Based on Cluster Event Size

Event Size Range	No. of Clusters	Percentage Retrieved(%)			
		SLCT	Loghound	Loghound-2	IPLoM
1 - 10	316	12.97	13.29	49.68	53.80
11 - 20	142	7.04	9.15	35.92	49.30
>21	68	15.15	16.67	77.27	51.52

The average IR F-Measure performance across the data sets, at this default support level, is 0.07, 0.04, 0.10, and 0.46 for SLCT, Loghound, Loghound-2, and IPLoM, respectively. However, as stated in [29], in cases where data sets have relatively long patterns or low minimum support thresholds are been used, a priori-based algorithms incur nontrivial computational cost during candidate generation. The event size statistics for our data sets are outlined in Table 3, which shows the *HPC* file as having the largest maximum and average event size. Loghound was unable to produce results on this data set with an absolute support value of 2, because the algorithm crashed due to the large number of item sets that had to be generated as can be seen in Fig. 10. This was however not a problem for SLCT (as it generates only 1 item sets). These results show that Loghound is vulnerable to the computational cost problems outlined in [29], which is however not a problem for IPLoM as its computational complexity is not adversely affected by long patterns or low minimum support thresholds. In terms of performance based on event size, Table 4 shows consistent performance from IPLoM irrespective of event size, while SLCT and Loghound seem to suffer for midsize clusters. Evaluations of Loghound considering the trailing variable problem shows Loghound achieving its best results for message types with a large event size and achieving results which are comparable to IPLoM in the other categories.

Another cardinal goal in the design of IPLoM is the ability to discover clusters in event logs irrespective of how frequently its instances appear in the data. The performance of the algorithms using this evaluation criterion is outlined in Table 5. The results show a reduction in performance for all the algorithms for clusters with a few instances; however, IPLoM's performance was more resilient.

The resource consumption results for SLCT, Loghound, and IPLoM are presented in Tables 6, 7, and 8. The tables

TABLE 5
Algorithm Performance Based on Cluster Instance Frequency

Instance Frequency Range	No. of Clusters	Percentage Retrieved(%)			
		SLCT	Loghound	Loghound-2	IPLoM
1 - 10	263	2.66	1.90	38.02	44.87
11 - 100	144	16.67	18.75	50.69	47.92
101 - 1000	68	20.59	23.53	63.24	72.06
>1000	51	34.00	38.00	74.00	82.00

TABLE 6
CPU Time in Minutes

	SLCT	Loghound	IPLoM
HPC	<1.0	3.5	2.0
Syslog	<1.0	<1.0	<1.0
Windows	<1.0	<1.0	<1.0
Access	<1.0	<1.0	5.4
Error	<1.0	<1.0	37.5
System	<1.0	<1.0	<1.0
Rewrite	<1.0	<1.0	<1.0

TABLE 7
Maximum Virtual Memory Consumption in KBs

	SLCT	Loghound	IPLoM
HPC	608,348	4,186,284	99,248
Syslog	600,284	600,284	98,992
Windows	601,232	601,056	98,992
Access	614,088	615,760	101,040
Error	623,504	613,964	101,040
System	599,636	600,032	98,992
Rewrite	600,032	600,032	98,992

show results for CPU, virtual memory, and resident memory consumption, respectively. We note that the results for Loghound on the HPC data set are results collected before its process crashed. However, the results of this case show why the process crashed as its virtual memory and resident memory consumption had gone up to $\sim$4 and $\sim$1.6 GB, respectively. This is for a file that is only about $\sim$11.4 MB on disk. These results corroborate our initial assertion on the failure of Loghound on this data set, i.e., a large number of item sets been generated at a low support value. The CPU consumption also does not show a true picture of time before the algorithm, the actual time between the start of the process and it crashing was 62 mins. The *ps* utility was run to only measure CPU time consumed by the process alone. In this case, all the CPU time spent by the OS in performing swaps between resident and virtual memory as result of the process were not recorded.

In spite of the fact that our implementation of IPLoM uses no sophisticated memory management techniques,[2] it produces results that are comparable to those of SLCT and Loghound in terms of both virtual memory and resident memory consumption. The memory consumption of IPLoM can still be improved by using optimized algorithms and data structures. In terms of CPU time, however its results are only comparable to those of SLCT and Loghound for the smaller data sets, i.e., Rewrite, Syslog, System, and Windows. We do not see this as a problem since IPLoM is designed for offline extraction of message types. Having said this, we believe there is still a lot of room for improvement in CPU time consumption for the following reasons:

- IPLoM is implemented in PHP while SLCT and Loghound were implemented in C. Implementing IPLoM in a low level language which can be compiled to object code will lead to faster processing times.
- The most important factor in determining the processing time for IPLoM is time spent scanning the database. This implies linear complexity for IPLoM. Our implementation of IPLoM was for research purposes, so each step of the algorithm was implemented separately. This implies that there was no information sharing between the steps of the algorithm. This led to unnecessary scans of the database for information gathering, which could have been avoided.
- IPLoMs partitioning of the database is in effect a decomposition of the message type extraction problem. This in effect makes IPLoM a good candidate for using parallel processing.

2. SLCT and Loghound utilize string hashing functions and cache trees for efficient memory utilization [12], [13].

4.3 Absolute Support Values

In this set of experiments, we compare the result of the algorithms using absolute support values in the range of 1-20, with a minimum of 2 for SLCT and Loghound. Our goal here, as with the experiments in Section 4.4, is to determine how varying the file support threshold within a low range of values affects the performance of the algorithms. As stated earlier, using low file support values ensures that we stand a good chance of finding all the message types in file, which is one of our goals.

The average F-Measure results using the “Micro,” “Macro,” and “IR” evaluations for SLCT, Loghound, and IPLoM are highlighted in Table 9. The results show that IPLoM performs better than the other algorithms on all data sets in the IR evaluation, which measures the goodness of the clusters produced. In the Micro- and Macroevaluations, IPLoM still does better than the other algorithms in general. However, we see performance improvement from SLCT and Loghound and in one case (with the Syslog data set) SLCT actually performs better than IPLoM.

TABLE 8
Maximum Resident Memory Consumption in KBs

	SLCT	Loghound	IPLoM
HPC	10,388	1,619,156	19,196
Syslog	1,260	1,260	19,008
Windows	2,660	2,976	19,072
Access	15,688	22,748	21,136
Error	24,900	20,968	21,056
System	320	820	18,988
Rewrite	920	1,252	19,084

TABLE 9
Average F-Measure Performance of Algorithms Using Absolute Support Values

F-MEASURE PERFORMANCE						
	HPC			Syslog		
	SLCT	Loghound	IPLoM	SLCT	Loghound	IPLoM
Micro	0.64	0.55	0.66	0.10	0.06	0.07
Macro	0.25	0.45	0.45	0.13	0.07	0.11
IR	0.02	0.01	0.59	0.14	0.08	0.14
	Windows			Access		
	SLCT	Loghound	IPLoM	SLCT	Loghound	IPLoM
Micro	0.22	0.25	0.28	0.00	0.00	0.00
Macro	0.17	0.18	0.22	0.11	0.15	0.20
IR	0.18	0.11	0.34	0.00	0.00	0.26
	Error			System		
	SLCT	Loghound	IPLoM	SLCT	Loghound	IPLoM
Micro	0.68	0.28	0.82	0.20	0.16	0.83
Macro	0.22	0.17	0.31	0.18	0.09	0.56
IR	0.01	0.01	0.43	0.15	0.07	0.75
	Rewrite					
	SLCT	Loghound	IPLoM			
Micro	0.08	0.05	0.83			
Macro	0.10	0.13	0.30			
IR	0.01	0.01	0.49			

The rest of the results are evaluated using the IR method. A detailed summary of the IR Recall, Precision, and F-Measure results can be found in Section B of the Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2011.138.

4.4 Percentage-Based Support Values

A system administrator can specify a support value using an absolute value (as in the section above) or a value that is dependent on the number of lines in the event log, i.e., a percentage. To determine IPLoMs performance according to this specification of support value, we ran another set of experiments using percentage-based support values. For the same reasons as described above, the range of values are low, i.e., 0.1-1.0 percent. The F-Measure results of this scenario show that IPLoM performs better than the other algorithms on all the tasks. A single factor ANOVA test performed at 5 percent significance on the results shows a statistically significant difference in all the results except in the case of the Syslog file. Detailed Recall, Precision, and F-Measure results for these experiments are given in Section A of the Appendix, available in the online supplemental material.

4.5 Parameter Sensitivity Analysis

IPLoM has five parameter values which can affect its results. These parameters are the File Support Threshold, Partition Support Threshold, Cluster Goodness Threshold, and the Upper Bound and Lower Bound (LB) thresholds used to decide if the "many" end of a 1-M relationship represents constant values or variable values. It is important that we assess the sensitivity of IPLoM's performance to the value settings of these parameters. In this section, we present such an analysis. We ran IPLoM against the data sets using a wide range of values as outlined in Table 10, because the FST used in IPLoM is similar to the support threshold used in SLCT and Loghound we also ran tests on them using the range of values for FST in Table 10.

TABLE 10
Parameter Value Ranges Used for Sensitivity Analysis

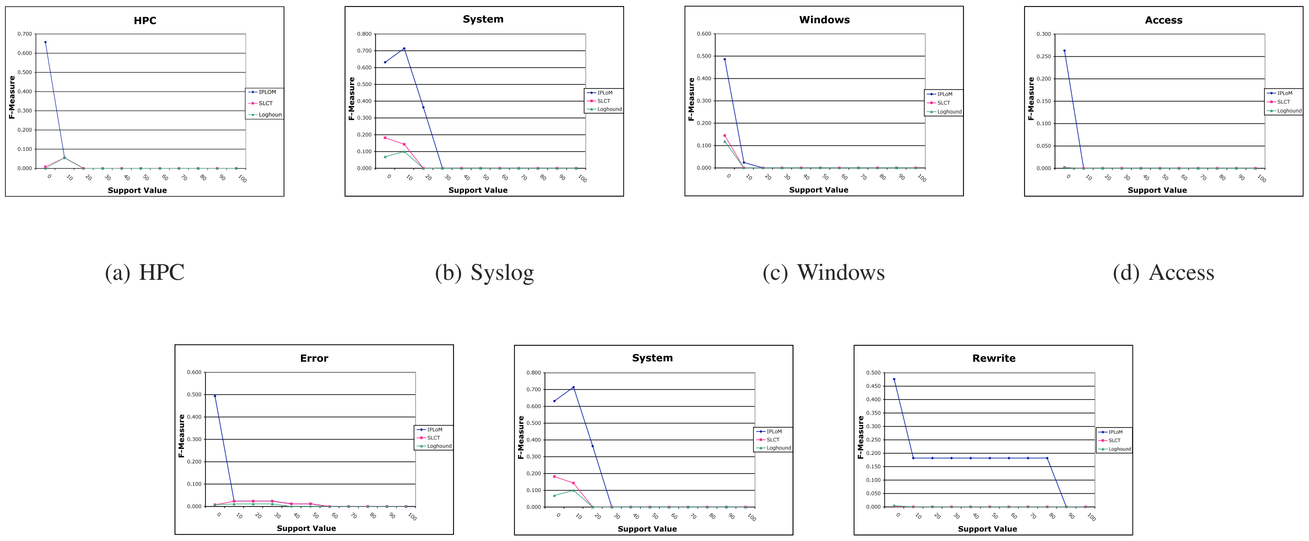
Parameter	Range
File Support Threshold(%)	0 - 100
Partition Support Threshold(%)	0 - 5
Lower Bound	0.1 - 0.5
Upper Bound	0.5 - 0.9
Cluster Goodness Threshold	0 - 1

The results show that IPLoM is most sensitive to varying values of FST as can be seen in Fig. 11. This can be explained by the observation that increasing the support value decreases the number of event types that can be found, since any event type with instances that fall below the support value cannot be found. The graphs however show that generally for support values greater than 20 percent there is not much difference in the performance of the algorithms. Using the standard deviation of the results over the range of results for each parameter, as seen in Table 11, we can evaluate the sensitivity of the algorithms to changing parameter values. The results show that IPLoM is stable in face of changing parameter values. The largest standard deviation values are found with IPLoM under the FST parameter, which is due to IPLoM's superior performance for FST values less than 20 percent.

4.6 Evaluating IPLoM Using BlueGene/L's (BGL) Logs

In this section, we present the evaluation results of clustering BlueGene/L's logs using IPLoM [30] and using the message types found in the analysis of the contents of the log. This section shows how IPLoM would scale to large and more complex data sets and demonstrates a practical application of the message types extracted.

The BlueGene/L data set is a publicly available high performance computing log data set [31]. The BlueGene/L supercomputer is a well-known HPC machine designed by IBM. It is located at Lawrence Livermore National Labs (LLNL) in Livermore, CA, USA. According to the Top-500 supercomputing site, BlueGene/L ranked #5 in its list of the fastest supercomputers in the world [32]. The BGL data contains $4.7M$ event log entries from BlueGene/L covering a 215 day period. The size of the event log file on disk is 1.2 GB. As with our other evaluations, message types were produced manually from the BGL data and we compared these to message types produced automatically by IPLoM. The same procedure used for manually discovering the message types for the other data sets was used in this case. IPLoM produced 332 message types on this data set compared with 394 produced manually. Other algorithms produced message types running into tens of thousands on this data [30]. The results of evaluating IPLoM's message types using our evaluation methodologies are outlined in Table 12. The



(a) HPC (b) Syslog (c) Windows (d) Access

(e) Error (f) System (g) Rewrite

Fig. 11. F-Measure performance of IPLoM, Loghound, and SLCT against FST values in the range 0-100 percent. Zero percent support values for SLCT and Loghound are equivalent to using an absolute support value of 2.

results show that IPLoM not only scales well to large data sets, but it can also achieve an F-Measure of 91 percent based on microaverage classification accuracy.

The IPLoM algorithm is a lightweight algorithm with linear complexity. The results highlighted in Fig. 12 show evaluations of running IPLoM against portions of the BGL data which differ in size by units of 100,000 events. The results show IPLoM to have linear complexity with regard to the size of an event log.

On the other hand, Nodeinfo is an alert (anomaly) detection method based on the entropy-based information content analysis of system logs [33]. Nodeinfo works based on the assumption that *"Similar computers correctly executing similar code should produce similar logs."* Nodeinfo does not fully capture message context as it does not use message types. Instead, it utilizes the concept of encoding token and token position pairs for dealing with unstructured messages. We applied the Nodeinfo algorithm to BGL data using the individual tokens in the log file as terms and again using the message types extracted by IPLoM as terms after message type transformation [34]. The results show that with this approach, we were able to process 100 times as much data per unit time without a drop in the anomaly detection accuracy of Nodeinfo.

TABLE 11
Standard Deviation over F-Measure Results for Parameter Values

	FST			PST	CGT	LB:UB
	Loghound	SLCT	IPLoM	IPLoM	IPLoM	IPLoM
HPC	0.017	0.017	0.197	0.009	0.107	0.048
Syslog	0.025	0.000	0.058	0.000	0.013	0.025
Windows	0.036	0.044	0.146	0.002	0.088	0.018
Access	0.000	0.001	0.079	0.036	0.085	0.006
Error	0.005	0.010	0.146	0.034	0.034	0.028
System	0.035	0.066	0.279	0.000	0.000	0.197
Rewrite	0.001	0.000	0.123	0.000	0.000	0.000

TABLE 12
IPLoM's Performance of BGL Data

	Recall	Precision	F-Measure
Micro	0.91	0.91	0.91
Macro	0.50	0.50	0.50
IR	0.50	0.60	0.54

4.7 Analysis of Performance Limitations

The IPLoM algorithm, as with all algorithms that utilize heuristics, is capable of making errors and does in fact make errors during its partitioning phase. Our analysis of these errors is described in this section. In some cases, these problems can be mitigated by preprocessing of the event data or post-processing of results, and we also describe the possible remedies.

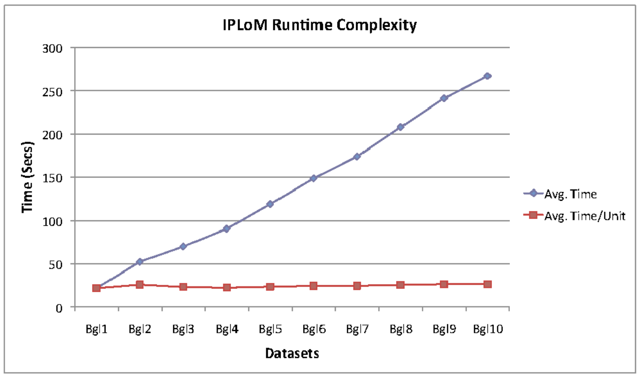
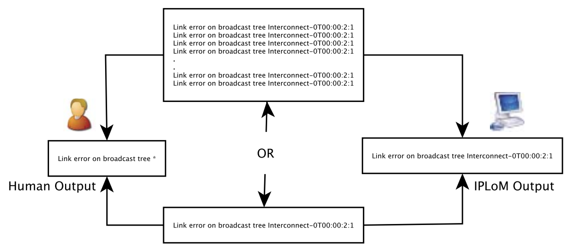


Fig. 12. Runtime complexity of IPLoM on the BGL data. Each Bgl$x$ data set contains $x * 100{,}000$ events.



```
Link error on broadcast tree Interconnect-0T00:00:2:1
Link error on broadcast tree Interconnect-0T00:00:2:1
Link error on broadcast tree Interconnect-0T00:00:2:1
Link error on broadcast tree Interconnect-0T00:00:2:1
.
.
Link error on broadcast tree Interconnect-0T00:00:2:1
Link error on broadcast tree Interconnect-0T00:00:2:1
```

```
Link error on broadcast tree *
```

```
Link error on broadcast tree Interconnect-0T00:00:2:1
```

```
Link error on broadcast tree Interconnect-0T00:00:2:1
```

Fig. 13. Example: insufficient information in data.

4.7.1 *Insufficient Information in Data*

Apart from the cluster descriptions produced by all the algorithms as output, IPLoM has the added advantage of producing the partitions of the log data which represent the actual clusters. This gives us two sets of results we can evaluate for IPLoM, the clusters and their descriptions. In our evaluation of the partition results of IPLoM, we discovered that in certain cases that it was impossible for IPLoM to produce the right cluster descriptions for a partition due to the fact that the partition contained only one event line or all the event lines were identical. This situation would not pose a problem for a human subject as they are able to use semantic and domain knowledge to determine the right cluster description. This problem is illustrated in Fig. 13. This indicates that the IR comparison of the cluster descriptions produced by IPLoM does not give a complete picture of IPLoM's performance. To get a complete picture of IPLoM's capabilities, we evaluated IPLoM's performance based on partitioning. These results are called *Partitions* in Fig. 14, while the cluster description results are called *Before*. The partition comparison differs from the cluster description by including as correct, cases where IPLoM came up with the right partition but was unable to come up with the right cluster description. The results show an average F-Measure of 0.48 and 0.78 for IPLoM when evaluating the results of IPLoM's cluster description output and partition output respectively. Similar results are also noticed for Precision and Recall.

Due to the fact that SLCT and Loghound do not generate partitions to evaluate against (these partitions can however be found through postprocessing if desired) and it can be argued that the insufficient information in data scenario could also apply to them, we constructed another experiment. In this case, we inserted counterexamples for all the cases where there was insufficient information in the event data for the algorithms to come up with the cluster descriptions and ran SLCT, Loghound, and IPLoM against the new data sets with counterexamples inserted. SLCT and Loghound were run in this case with the absolute support values which gave their best results in the experiments described in Section 4.3 above while IPLoM was run in its default state. This results are called *After* in Fig. 14. The results show that unlike SLCT and Loghound, IPLoM was able to make use of the new information to improve its results in all cases. However, we consider the *Partitions* results to be the most accurate illustration of IPLoM's capabilities. These results show that IPLoM can achieve an average Recall of 0.83, Precision of 0.74, and F-Measure of 0.78.

4.7.2 *Ambiguous Token Delimiters*

In the problem of message type extraction, the assumption is that the space character acts as the token delimiter. On close inspection of certain messages in the log files, we find this not to be true in all cases. The most common example occurs when part or all of a message contains a "$variable = value$" phrase. In some cases, there's no space character between the *variable* token and the $=$ sign and also between the *value* token and the $=$ sign. This scenario becomes a problem for IPLoM when for instance these log messages "*Temperature reading: ambient* $= 30$," "$Temperature\ reading: ambient = 25$," and "$Temperature\ reading: ambient = 28$ are evaluated to the type "$Temperature\ reading: ambient = *$" by a human observer. When and if IPLoM correctly produces a partition

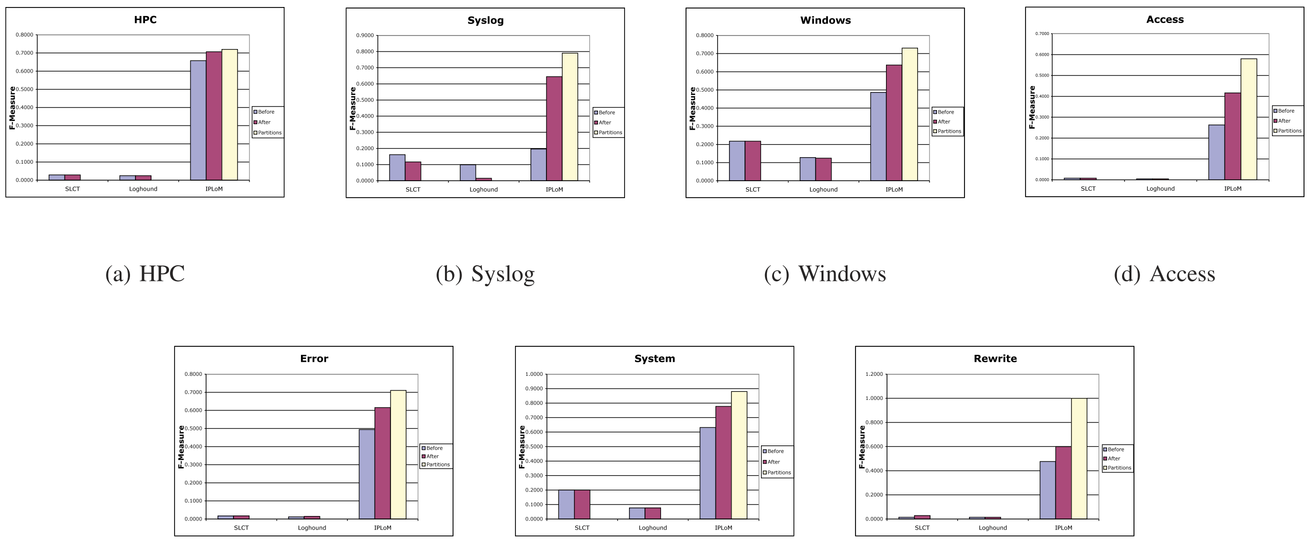


Fig. 14. Comparing F-Measure performance of IPLoM, Loghound, and SLCT before insertion of counterexamples, after insertion of counterexamples and evaluating the accuracy of cluster partitioning before the insertion of counterexamples.

containing, these log messages the type produced will be *"Temperature reading: *,"* due to IPLoM's inability to separate the tokens in the "$variable = value$" phrase. An approach to mitigating this problem is outlined in [7].

4.7.3 Clusters with Events of Variable Size

Another scenario that occurs is with clusters with events of variable size. We assume that messages belonging to the same cluster should have the same number of tokens or event size. Again on close inspection, we find this not to be true in some cases. Clusters with events of variable sizes usually occur when a variable position in the line format can contain strings instead just single words. For example, consider these messages *"The LightScribe service has started"* and *"The Message Queuing service has started"* which should belong to the same message type and have message type description *"The * service has started,"* having a differing number of tokens. These messages would be separated by the Step 1 of IPLoM's partitioning process and IPLoM would likely produce two message type descriptions for this cluster *"The * service has started"* and *"The * * service has started,"* the latter line format being redundant.

This problem can be mitigated by performing message type description refinement after the message types are produced. An approach like the string edit distance used in [10] can be utilized for this step if necessary.

5 Conclusion and Future Work

Due to the size and complexity of sources of information used by system administrators in fault management, it has become imperative to find ways to manage these sources of information automatically. Application logs are one such source. We present our work on designing a novel algorithm for message type extraction from log files, IPLoM. So far, there is no standard approach to tackling this problem in the literature [9]. Message types are semantic groupings of system log messages. They are important to system administrators, as they aid their understanding of the contents of log files. Administrators become familiar with message types over time and through experience. Our work provides a way of finding these message types automatically. In conjunction with the other fields in an event (host names, severity), message types can be used for more detailed analysis of log files.

Through a 3-step hierarchical partitioning process, IPLoM partitions log data into its respective clusters. In its fourth and final stage, IPLoM produces message type descriptions or line formats for each of the clusters produced. IPLoM is able to find message clusters whether or not its instances are frequent. We demonstrate that IPLoM produces cluster descriptions, which match human judgement more closely when compared to SLCT, Loghound, and Teiresias. It is also shown that IPLoM demonstrated statistically significantly better performance than either SLCT or Loghound on six of the seven different data sets tested. These results however do not imply that SLCT and Loghound are not useful tools for event log analysis. They can still be useful for log analysis that involves other fields in an event. However, our results show that a specialized algorithm such as IPLoM can significantly improve the abstraction level of the unstructured message types extracted from the data.

Message types are fundamental units in any application log file. Determining what message types can be produced by an application accurately and efficiently is therefore a fundamental step in the automatic analysis of log files. Message types, once determined, not only provide groupings for categorizing and summarizing log data, which simplifies further processing steps like visualization or mathematical modeling, but also a way of labeling the individual terms (distinct word and position pairs) in the data.

Future work on IPLoM will involve using the information derived from the results of IPLoM in other automatic log analysis tasks which help with fault management. We also intend to implement an optimized version of IPLoM in a low level programming language such as C/C++, and make it publicly available on our website.[3] Lastly, our future work will continue on the integration of machine learning techniques and information retrieval with message type clustering in order to study automation of fault management and troubleshooting for computer systems.

Acknowledgments

This research is supported by a Natural Science and Engineering Research Council of Canada (NSERC) Strategic Project Grant. The authors would also like to acknowledge the staff of Palomino System Innovations, Inc., TARA, Inc., and Dal-CS Tech-Support for their support in completing this work. This work is conducted as part of the Dalhousie NIMS Lab at http://www.cs.dal.ca/projectx/.

References

[1] J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer,* vol. 36, no. 1, pp. 41-50, Jan. 2003.

[2] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, Indexing, Clustering, and Retrieving System History," *Proc. 20th ACM Symp. Operating Systems Principles,* pp. 105-118, 2005.

[3] M. Jiang, M.A. Munawar, T. Reidemeister, and P.A. Ward, "Dependency-Aware Fault Diagnosis with Metric-Correlation Models in Enterprise Software Systems," *Proc. Sixth Int'l Conf. Network and Service Management,* pp. 137-141, 2010.

[4] M. Klemettinen, "A Knowledge Discovery Methodology for Telecommunications Network Alarm Databases," PhD dissertation, Univ. of Helsinki, 1999.

[5] S. Ma and J. Hellerstein, "Mining Partially Periodic Event Patterns with Unknown Periods," *Proc. 16th Int'l Conf. Data Eng.,* pp. 205-214, 2000.

[6] Q. Zheng, K. Xu, W. Lv, and S. Ma, "Intelligent Search for Correlated Alarm from Database Containing Noise Data," *Proc. Eighth IEEE/IFIP Network Operations and Management Symp.,* pp. 405-419, 2002.

[7] J. Stearley, "Towards Informatic Analysis of Syslogs," *Proc. IEEE Int'l Conf. Cluster Computing,* pp. 309-318, 2004.

[8] A. Makanju, A.N. Zincir-Heywood, and E.E. Milios, "Storage and Retrieval of System Log Events Using a Structured Schema Based on Message Type Transformation," *Proc. 26th ACM Symp. Applied Computing (SAC),* pp. 525-531, Mar. 2011.

[9] W. Xu, L. Huang, A. Fox, D. Patterson, and M.I. Jordan, "Detecting Large-Scale System Problems by Mining Console Logs," *SOSP '09: Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles,* pp. 117-132, 2009.

[10] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," *Proc. Ninth IEEE Int'l Conf. Data Mining (ICDM '09),* pp. 149-158, Dec. 2009.

3. http://web.cs.dal.ca/~makanju/iplom.

[11] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. 20th Int'l Conf. Very Large Data Bases (VLDB)*, J.B. Bocca, M. Jarke, and C. Zaniolo, eds., pp. 487-499, 1994.

[12] R. Vaarandi, "A Data Clustering Algorithm for Mining Patterns from Event Logs," *Proc. IEEE Workshop IP Operations and Management*, pp. 119-126, 2003.

[13] R. Vaarandi, "A Breadth-First Algorithm for Mining Frequent Patterns from Event Logs," *Proc. IFIP Int'l Conf. Intelligence in Comm. Systems*, vol. 3283, pp. 293-308, 2004.

[14] I. Rigoutsos and A. Floratos, "Combinatorial Pattern Discovery in Biological Sequences: The TEIRESIAS Algorithm," *Bioinformatics*, vol. 14, pp. 55-67, 1998.

[15] C. Lonvick, "The BSD Syslog Protocol," RFC3164, Aug. 2001.

[16] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic Subspace Clustering of High Dimensional for Data Mining Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 1998.

[17] S. Guha, R. Rastogi, and K. Shim, "CURE: An Efficient Clustering Algorithm for Large Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 73-84, 1998.

[18] S. Goil, H. Nagesh, and A. Choudhary, "MAFIA: Efficient and Scalable Subspace Clustering for Very Large Data Sets," technical report, Northwestern Univ., 1999.

[19] J.H. Bellec and M.T. Kechadi, "CUFRES: Clustering using Fuzzy Representative Event Selection for the Fault Recognition Problem in Telecommunications Networks," *Proc. ACM First PhD Workshop in CIKM*, pp. 55-62, 2007.

[20] T. Li, F. Liang, S. Ma, and W. Peng, "An Integrated Framework on Mining Log Files for Computing System Management," *Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery in Data Mining (KDD '05)*, pp. 776-781, 2005.

[21] B. Topol, "Automating Problem Determination: A First Step Toward Self Healing Computing Systems," IBM White Paper, http://www-106.ibm.com/developerworks/autonomic/library/ac-summary/ac-prob.html, Oct. 2003.

[22] J. Stearley, "Sisyphus Log Data Mining Toolkit," http://www.cs.sandia.gov/sisyphus, Jan. 2009.

[23] A. Makanju, S. Brooks, N. Zincir-Heywood, and E.E. Milios, "Logview: Visualizing Event Log Clusters," *Proc. Sixth Ann. Conf. Privacy, Security and Trust (PST)*, pp. 99-108, Oct. 2008.

[24] F. Salfener and M. Malek, "Using Hidden Semi-Markov Models for Effective Online Failure Prediction," *Proc. 26th IEEE Int'l Symp. Reliable Distributed Systems*, pp. 161-174, 2007.

[25] G. Grabarnik, A. Salahshour, B. Subramanian, and S. Ma, "Generic Adapter Logging Toolkit," *Proc. Int'l Conf. Autonomic Computing*, pp. 308-309, 2004.

[26] W. van der Aalst and H. Verbeek, "Process Mining in Web Services: The Websphere Case," *IEEE Bull. of the Technical Committee on Data Eng.*, vol. 31, no. 3, pp. 45-48, 2008.

[27] W.D. Pauw, M. Lei, E. Pring, L. Villard, M. Arnold, and J. Morar, "Web Services Navigator: Visualizing the Execution of Web Services," *IBM Systems J.*, vol. 44, no. 4, pp. 821-845, 2005.

[28] Los Alamos Nat'l Security LLC, "Operational Data to Support and Enable Computer Science Research," http://www.pdl.cmu.edu/FailureData/ and http://institutes.lanl.gov/data/fdata/, Jan. 2009.

[29] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 1-12, 2000.

[30] A. Makanju, A.N. Zincir-Heywood, and E.E. Milios, "Extracting Message Types from BlueGene/L's Logs," *Proc. 22nd ACM Symp. Operating Systems Principles, Workshop the Analysis of System Logs (WASL '09)*, 2009.

[31] "Usenix - the Computer Failure Data Repository," http://cfdr.usenix.org/data.html, June 2009.

[32] TOP500.Org, "Top500 Supercomputing Sites," http://www.top500.org/, June 2009.

[33] A. Oliner, A. Aiken, and J. Stearley, "Alert Detection in System Logs," *Proc. Int'l Conf. Data Mining (ICDM)*, pp. 959-964, 2008.

[34] A. Makanju, A.N. Zincir-Heywood, and E.E. Milios, "Fast Entropy Based Alert Detection in Supercomputer Logs," *Proc. DSN Workshop Proactive Failure Avoidance, Recovery and Maintenance*, 2010.

Adetokunbo Makanju received the MSC degree in computer science from Dalhousie University, Halifax, NS, Canada in 2007. He is currently working toward the PhD degree at Dalhousie University. His research interests include but are not limited to the areas of event log analysis, wireless networks, intrusion detection, and genetic programming. In particular, his focus is on the applications of machine intelligence to real-world problems. He is a member of the IEEE.

A. Nur Zincir-Heywood received the PhD degree in network information retrieval from the Department of Computer Engineering, Ege University, Turkey, in 1998. Currently, she is a professor with the Faculty of Computer Science, Dalhousie University, Halifax, NS, Canada. Her research interests include autonomic network systems in the areas of network management and network information classification. She is a member of the IEEE and the ACM.

Evangelos E. Milios received the diploma in electrical engineering from the National Technical University of Athens, Greece, in 1980 and the master's and PhD degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, from where he graduated in 1986. Since July of 1998, he has been with the Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, where he is a professor and Killam Chair of computer science. He served as a member of the ACM Dissertation Award committee (1990-1992), a member of the AAAI/SIGART Doctoral Consortium Committee (1997-2001) and he is co-editor-in-chief of the journal *Computational Intelligence*. His current research activity is centered on modeling and mining of content and link structure of networked information spaces. He is a senior member of the IEEE.