Contextual Analysis of Program Logs for Understanding System Behaviors

Qiang Fu, Jian-Guang Lou, Qingwei Lin, Rui Ding, Dongmei Zhang
Microsoft Research Asia
Beijing, China
{qifu, jlou, qlin, juding, dongmeiz}@microsoft.com

Tao Xie
Department of Computer Science
North Carolina State University
NC, USA
xie@csc.ncsu.edu

***Abstract*—Understanding the behaviors of a software system is very important for performing daily system maintenance tasks. In practice, one way to gain knowledge about the runtime behavior of a system is to manually analyze system logs collected during the system executions. With the increasing scale and complexity of software systems, it has become challenging for system operators to manually analyze system logs. To address these challenges, in this paper, we propose a new approach for contextual analysis of system logs for understanding a system's behaviors. In particular, we first use execution patterns to represent execution structures reflected by a sequence of system logs, and propose an algorithm to mine execution patterns from the program logs. The mined execution patterns correspond to different execution paths of the system. Based on these execution patterns, our approach further learns essential contextual factors (e.g., the occurrences of specific program logs with specific parameter values) that cause a specific branch or path to be executed by the system. The mining and learning results can help system operators to understand a software system's runtime execution logic and behaviors during various tasks such as system problem diagnosis. We demonstrate the feasibility of our approach upon two real-world software systems (Hadoop and Ethereal).**

***Index Terms*—Contextual Analysis, understanding system behaviors, log analysis.**

I. INTRODUCTION

With the increasing scale and complexity of software systems, it has become more and more difficult for system operators to understand the behaviors of software systems for tasks such as system problem diagnosis. For example, system operators need to understand system behaviors to figure out why a software system is in the current status. With such an understanding, they can choose the right operations to achieve the desired goal. System behaviors include a series of actions executed by the system and the corresponding changes in the system states. Although operators usually investigate a system starting from a specific state of interest, e.g., a hang state or failure state, contextual information for reaching that state is critical for identifying why the system runs in that state. Such contextual information includes how previous actions are executed by the system, what the historical system states are before running into the state of interest, what the input data is, etc.

To help system operators understand a system's behaviors, three main sources of information are available: system documentations, source code, and program logs. Ideally, system documentations are expected to provide detailed and up-to-date information about the system. However, in practice, system documentations are often incomplete, outdated, or even unavailable. Many systems are not well documented due to tight release schedules or poor project management. Program source code is another source that provides accurate and detailed information about a system. However, the source code of the system is not always available to system operators. For example, in many cases, a system may include third-party components whose source code is not available.

The information provided by program logs [1] has been the most widely used source for system operators to understand system behaviors, especially for a large-scale online system. Program logs contain a wealth of information to help manage systems. Most systems print out program logs during their executions to record system runtime actions and states that can directly reflect system runtime behaviors. System operators usually use these logs to track a system to detect and diagnose system anomalies.

However, there are challenges in analyzing program logs in order to understand a system's behaviors. The program logic of a system usually has a lot of branches, and thus the system's behaviors may be quite different under different input data or environmental conditions. Knowing the execution behavior under different inputs or configurations can greatly help system operators to understand system behaviors. However, there may be a large number of different combinations of inputs or parameters under different system behaviors. Such complexity poses difficulties for analyzing contextual information related to the state of interest.

To address this challenge, in this paper, we propose a new approach for the contextual analysis of program logs to better understand a system's behaviors. In particular, we use execution patterns to represent the execution structures reflected by program logs, and propose an algorithm to mine execution patterns from the program logs. The mined execution patterns correspond to different execution paths of the system. Based on the execution patterns, our approach further learns the essential contextual factors (e.g., occurrences

of specific program logs with specific parameter values) that cause a specific branch or path to be executed by the system. The mining and learning results can help system operators understand the execution logic and behaviors of a software system for their various maintenance tasks such as system problem diagnosis.

This paper makes the following research contributions:

- We conduct Formal Concept Analysis (FCA) [4] to analyze log messages, and construct a concept lattice graph. The learned graph is used to mine execution patterns and to model relationships among different execution patterns. Such relationships represent branch structures in the program execution logic.
- Based on the lattice graph, we propose a feature extraction technique and use decision trees to learn branch conditions. The learned branch conditions reveal essential contextual factors that determine which code branches the system will take at bifurcation points.

II. BACKGROUND

Developers often write log-printing statements [1] in key points of program source code to track system actions and states during system execution. Table 1 lists two examples of log-printing statements and the corresponding log messages in Hadoop.

TABLE I. EXAMPLES OF LOG-PRINTING STATEMENTS AND LOG MESSAGES

Log-printing statement	Log Message	Index
LOG.info("JVM with ID: " + jvmId + " given task: " + tip.getTask().getTaskID())	JVM with ID: jvm_200906291359_0008_r_1815559152 given task: attempt_200906291359_0008_r_000009_0	161
LOG.info("Adding task '" + taskid + "' to tip " + tip.getTIPId() + ", for tracker '" + taskTracker + "'")	Adding task 'attempt_200906291359_0008_r_000009_0' to tip task_200906291359_0008_r_000009, for tracker 'tracker_msramcom-pt5.fareast.corp.microsoft.com:127.0.0.1/127.0.0.1:1505'	73

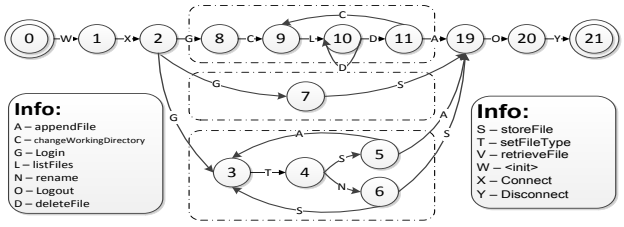

Fig. 1. A part of CVS workflow

Generally, each log message consists of two different types of content: a constant string and parameter values. For example, for the first log-printing statement in Table 1, the constant string is "JVM with ID: given task:", and the printed parameters are JvmId and TaskId. The log messages printed by the same log-print statement contain the same constant string, and are considered to be of the same type, represented by the constant string. For the first log message in Table 1, its message type signature is "JVM with ID:~ given task:~", where "~" means a parameter place holder. Its parameter values are "jvm_200906291359_0008 _r_1815559152" and "attempt_200906291359_0008_r_000 009_0" respectively.

Free-form texts in log messages can be converted to a structured representation [2]. The structured representation of each log message is a tuple consisting of a timestamp, a message type, and a parameter value list: <timestamp, message type, param1-value, param2-value, … paramN-value>. For convenience, we give each message type a unique index. The indexes of message types in Table 1 are 161 and 73, respectively. A printed parameter can be uniquely identified by a message type and a position index (messageTypeIndex, positionIndex). For example, (73,1) represents the first parameter in messages of type 73, and (161,2) represents the second parameter in messages of type 161. We denote the message type that parameter α belongs to as L(α). For a log message m of type L(α), the value of parameter α in m is denoted as v(α,m). For example, the value of parameter (73,1) in the second log message in Table 1 is attempt_200906291359 _0008_r_000009_0. All distinct values of parameter α in all log messages of type L(α) form a value set of α, denoted as V(α).

Our approach accepts as input a number of log-message groups, each of which consists of a log-message sequence corresponding to an execution instance related to the same object identifier (e.g., JobID and TransactionID), which is reflected by specific parameter values in different log messages. Previous approaches [2] can be extended to construct such log-message groups from raw log messages.

TABLE II. EXAMPLES OF LOG-MESSAGE GROUPS, EACH CORRESPONDING TO THE SAME TRANSACTION TYPE

Transaction index	Sequence of log messages
1	W X G T S A O Y
2	W X G T N S O Y
3	W X G T N S T S A O Y
4	W X G S O Y
5	W X G C I D A O Y

TABLE III. CONTEXT OF EXECUTIONS IN TABLE 2

	W	X	G	O	Y	S	T	N	A	C	I	D
1	○	○	○	○	○	○	○		○			
2	○	○	○	○	○	○	○	○				
3	○	○	○	○	○	○	○	○	○			
4	○	○	○	○	○	○						
5	○	○	○	○	○				○	○	○	○

For example, Table 2 lists example log-message groups, each of which consists of a log-message sequence representing the execution flow for serving the same transaction (each log message is represented with its type in the table). Such examples of log-message groups are constructed from raw log messages that are the result of running a Concurrent Versions System (CVS) system whose partial workflow is shown in Figure 1 (presented by Lo et al. [3]). The system execution goes into the five different code paths for serving five types of transactions.

III. APPROACH

To learn essential contextual factors that cause a specific branch or path to be executed by the system, our approach consists of two major steps. First, given the log-message groups, we apply Formal Concept Analysis (FCA) [4] to identify execution patterns and build up the lattice graph to model the relationships among execution patterns (Section 3.1). Second, from the constructed lattice graph, we extract features from log messages, and apply machine learning techniques to learn essential contextual factors that cause a specific branch or path to be executed by the system (Section 3.2). The learned essential contextual factors can provide useful information for understanding why the system exhibits specific behaviors.

A. Mining Execution Patterns and Relationships

Given log-message groups, we apply Formal Concept Analysis (FCA) [4] to identify execution patterns and build up the lattice graph to model the relationships among execution patterns. In FCA, given a context $I=(OS, AS, R)$, consisting of a binary relationship R between objects (from the set OS) and attributes (from the set AS), a concept c is produced by FCA as a pair of sets (X, Y) such that

$$X = \{o \in OS | \forall a \in Y : (o, a) \in R\}$$
$$Y = \{a \in AS | \forall o \in X : (o, a) \in R\}$$

Here, X is called as the extent of the concept c and Y is its intent. Concepts are ordered by their partial relationship (noted as $\leq_R$). Such partial ordering relationships can induce a complete lattice on concepts, called the lattice graph.

As an example, Table 3 is a typical context representation that defines the binary relationship between the objects and attributes: operations executed by different code paths according to Table 2. Here, the objects (from rows) are execution code paths (represented by log-message sequences), and the attributes (from columns) are operations (represented by individual log messages). A circle in a table cell represents the situation in which the corresponding object has the corresponding attribute. With the context illustrated in Table 3, we apply FCA to obtain the lattice graph shown in Figure 2.

In Figure 2, each node contains an execution pattern (i.e., the message-type set in the second line) and transaction indexes (i.e., the indexes of transactions in the first line) whose produced messages contain the message types in the execution pattern. For a node, its parent (a connected upper one node) node's intent represents a trunk segment of code paths. Each transaction index appears in the sequence (from top to down) of the connected nodes' extents. The corresponding series of intents of these connected nodes indicates the series of execution patterns, which illustrate the system execution from the trunk to the more and more specific branch for serving the transaction.

Although other kinds of information, e.g., the temporal-order information and the message-count information, may also facilitate system understanding (e.g., based on frequent sequence mining or automaton learning), our FCA mainly focuses on the set of distinct message types to model system-execution code paths. The rationale is that using the message-type set to represent different code paths is robust in the presence of message disordering, which is very common in distributed systems because clocks of different machines are hard to ideally synchronize.

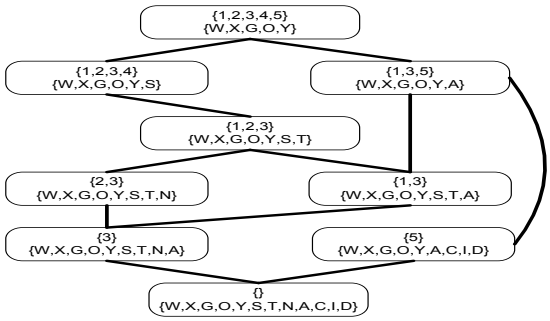

Fig. 2. Topology of execution patterns as a lattice graph

B. Learning Essential Contextual Factors

In practice, a system selects its execution code paths depending on the values of the input argument variables or runtime system states. In general, branch conditions can be various kinds of predicates and expressions, which can be either numerical values or simple enumerable data (e.g., system states and return codes). Developers of a system often record the values of such conditional variables as log parameters for diagnosing problems and tracking object life cycles. For example, a web application may execute different code paths according to the different HTTP request types, e.g., get or post. Such key states are recorded in logs.

To discover conditional predicates that determine the selection of specific execution branches or paths, from the lattice graph constructed by FCA, we extract features from log messages and apply machine learning techniques to learn essential contextual factors that cause a specific branch or path to be executed by the system.

In particular, we formalize this problem as a supervised classification problem. For each node c in the lattice graph, we denote its child concept nodes as c_1, c_2, …, c_n. In order to determine why the code path of c would branch to the code paths of its child concepts $c_i (1 \leq i \leq n)$, we learn n classifiers for n node pairs (c, c_1), …, (c, c_n). The i^{th} classifier uses the logged parameter values in log messages produced by transactions within $O(c)$ to predict whether a transaction within $O(c)$ (i.e., the extent of concept c) would branch to the code path of c_i. In other words, for a transaction in $O(c)$, the i^{th} classifier predicts whether the transaction is contained in $O(c_i)$ according to logged parameter values in log messages of the transaction.

During the labeling, each transaction is given a class label depending on its appearance in the child nodes' extents. At the same time, features are extracted from the enumerable values of system state parameters printed in the log messages of the transaction. We mainly focus on the enumerable system state parameters, because each of the state parameters has only a constantly small set of values. We plan to extend our algorithm to handle real-type system variables in our future research.

After labeling and feature extraction (i.e., constructing feature vectors), we obtain a set of feature vectors with

corresponding labels (i.e., +1 or -1) for each pair of nodes in the lattice graph. We feed such labeled feature vectors as training data to the C4.5 decision tree learning algorithm to learn a decision tree for the pair of parent-child nodes. A decision tree is a classical classification model that can provide intuitive cues for operators to understand how each feature dimension is correlated to the class labels. In our scenarios, the learned decision tree can clearly tell us how a specific combination of the values of state variables determines that the system executes the code branch of the child node. Therefore, the learned classifier reveals the essential factors causing branches. The illustrative example is shown in the next section.

IV. Preliminary Results

We applied our approach to two open source systems: Ethereal and Hadoop. Our test bed for Hadoop (version 0.19) contained 16 machines connected with a 1G Ethernet switch. We ran some sample applications, such as WordCount and Sort, and collected the resulting log data (24 million lines). In the original source code of Ethereal, there is no log-printing statement. To collect logs, we added log-printing statements at the entry points of all functions related to dealing with the following protocols: IPv6, IP, arp, http, ftp, nbns, icmpv6m, tcp, udp, and ethernet2. We ran Ethereal to collect data packs from a LAN (about 500 machines) for about one hour. The produced log data from the two systems were processed by an extension of our previous method [2] to supply log-message groups to our approach as input.

We first used FCA to construct a lattice graph from the given log-message groups, and then learned the essential contextual factors that determine a branch code path executed by the system. In particular, we applied the technique discussed in Section 3-B for each pair of parent-child nodes in the lattice graph. We used a 4-fold cross-validation method to evaluate the accuracy of our learned decision trees. For a node c, every transaction in its extent O(c) is a sample that has an associated feature vector and a corresponding label. We randomly partitioned all samples (i.e., all transactions in O(c)) into four subsamples, used three of them as the training set to learn a decision tree, and then test the learned decision tree on the remaining subsample to measure the accuracy. We repeated the procedure four times. Each time, we used a different subsample as the testing set and the remaining subsamples as the training set, and obtained the accuracy for the learned decision tree. Finally, we used the average accuracy as the accuracy of learning contextual factors for this pair of parent-child nodes.

In our evaluation, we applied the learning algorithms on only the node pairs in which the child node's extent contains more than 100 elements, because a small number of samples usually cannot provide results of statistical significance. We also excluded another kind of node. In these nodes, log-message groups of different transactions contain the identical log messages. In other words, message groups corresponding to different transactions do not contain discriminative information to indicate why the system runs into different branches. Therefore, for such nodes, it is impossible to learn the essential contextual factors from logs. We call such nodes trivial nodes. These trivial nodes indicate that some important information is missing in the log data. Such feedback provides guidance for developers to log additional necessary information in log-printing statements to better perform log-based monitoring, diagnosis, or understanding of system behaviors.

Hadoop Results. In Hadoop, there are different kinds of object identifiers, which define different kinds of transactions. For example, when we investigate system behaviors for serving tasks, each task is considered a transaction. Accordingly, log messages with the same Task ID are grouped together as the message group of a transaction. We performed the proposed approach to understand system behaviors with specified object identifiers, i.e., Block ID, Task ID, Attempt ID, and JVM ID, respectively. The results are shown in Table 4.

TABLE IV. Average accuracy of learned decision trees on Hadoop

Identifier	Node	Node-L	Node-NE	Accuracy
Attempt ID	109	69	58	94.60%
Block ID	43	4	2	97.16%
Task ID	5	4	4	96.87%
JVM ID	5	4	0	--

In the table, Column Node shows the number of nodes in the lattice graph. Column Node-L shows the number of nodes (denoted as Node-L) whose extents contain more than 100 elements. Column Node-NE shows the number of non-trivial nodes (denoted as Node-NE) in Node-L. For each pair of parent-child nodes in which the nodes are in Node-NE, we learned the decision tree whose accuracy is evaluated with 4-fold cross-validation. The average accuracy of all learned decision trees on the lattice graph is listed in Column Accuracy.

Ethereal Results. The only object identifier of interest is Packet ID. Therefore, log messages with the same Packet ID form the message group of a transaction. The analysis results are listed in Table 5.

TABLE V. Average accuracy of learned decision trees on Ethereal

Identifier	Node	Node-L	Node-NE	Accuracy
Packet ID	28	16	14	99.95%

Result Analysis. By manually inspecting the learned classifiers for both Hadoop and Ethereal, we found that the classifiers not only have high accuracy for determining the execution branches, but also offer meaningful interpretation of system behaviors.

References

[1] D. Yuan, S. Park and Y. Zhou. Characterising logging practices in open-source software. In *Proc. ICSE'12*, 2012.

[2] Q. Fu, J. Lou, Y. Wang and J. Li, Execution anomaly detection in distributed systems through unstructured log analysis, In *Proc. ICDM' 09*, pp 149-158, 2009.

[3] D. Lo, L. Mariani, and M. Pezzè, Automatic steering of behavioral model inference, In *Proc. ESEC/FSE' 09*, 2009.

[4] B. Ganter and R. Wille, Formal concept analysis, Springer-Verlag, Berlin, Heidelberg, New York, 1996.