Log Filtering and Interpretation for Root Cause Analysis

Hamzeh Zawawy
University of Waterloo
Waterloo, Canada
hzawawy@engmail.uwaterloo.ca

Kostas Kontogiannis
National Technical University of Athens
Athens, Greece
kkontog@softlab.ntua.gr

John Mylopoulos
University of Toronto
Toronto, Canada
jm@cs.toronto.edu

***Abstract*—Problem diagnosis in large software systems is a challenging and complex task. The sheer complexity and size of the logged data make it often difficult for human operators and administrators to perform problem diagnosis and root cause analysis. A challenge in this area is to provide the necessary means, tools, and techniques for the operators to focus their attention to specific parts of the logged data reducing thus the complexity of the diagnostic process. In this paper, we propose a framework for filtering logs according to specific analysis goals and diagnostic hypotheses set by the user or by an automated process. More specifically, the proposed framework uses annotated goal trees to model the constraints and the conditions by which the functionality of a particular system is being delivered. Next, a transformation process maps such constraints and conditions to a collection of queries that can be either applied to a relational database that stores the logged data or use Latent Semantic Indexing to identify the most relevant log entries for the given query. The results of such queries provide a subset of the logged data that is compliant with the goal tree and can be used by a diagnostic SAT-solver based algorithm. Experimental results show that the filtering process can reduce the time and complexity of the diagnosis when applied to multi-tier heterogeneous service oriented systems.**

***Index Terms*—Log analysis, root cause analysis, goal model, performance, latent semantic indexing.**

I. Introduction

Root cause analysis (RCA hereafter) pertains to a set of techniques and processes that aim to discover faults that produce an observed failure. A common approach to RCA is to analyze log files and identify deviations from normal or expected behavior. These deviations are then examined so that potential causes can be identified. RCA performance is limited by the size of the logs considered. This problem becomes more apparent when large software systems are concerned. These systems are comprised of many components each of which may have different logging and monitoring mechanism so that, the logged data may often be too many and too complex for a human operator to analyze. In this respect, the problem becomes to reduce the size of the logged data by filtering out data that are not relevant to a particular diagnostic hypothesis, and to focus the attention of the human operators to events that may relate to root causes of an observed failure.

This paper presents a framework that helps on the analysis of streaming log data and interprets a selected subset of this data for the purpose of RCA and system diagnostics in service oriented systems. This framework provides a reduced log data set feeding into an existing root cause identification component based on a SAT-solver algorithm [14]. More specifically, the log reduction framework uses goal models that can be extracted either from the system analysts or by reverse engineering the source code [16]. In this paper, we propose to annotate goal models with *precondition*, *occurrence*, and *effect* predicates that can be used either to generate SQL queries to be applied against the log data pool or can be used for identifying related log entries by using Latent Semantic Indexing (LSI hereafter). In this respect, when the user experiences a system failure, the corresponding goal models and their annotations are used to generate queries that yield reduced log data that provide an initial focus to the human operator towards identifying the root cause of the problem being observed. To evaluate the proposed framework, we have applied it to a service oriented system we have built using a set of commercial-off-the-shelf products.

This paper is structured as follow. Section II presents related research work. Section III provides a description of the components and algorithms of the proposed extended monitoring framework. Section IV describes the normalization process applied on the natively generated log data. Details about the two approaches taken for log reduction are described in Section V. Section VI describes the use case scenario used and the corresponding test environment. Section VII describes the empirical evaluation of the framework. Section VIII summarizes the contributions and conclusions.

II. Related Work

This paper proposes a framework for log reduction and interpretation by limiting the size of the log data that need to be considered for verifying or denying a diagnosis in the context of a distributed environment. To model business processes, applications, services, etc., we use annotated goal models that have been formalized in Giorgini et al. [4]. A similar approach was used by [1] to monitor and ensure that service oriented systems satisfy their requirements by using BPEL based models annotated with logical predicates. Other approaches for understanding and monitoring the behavior of enterprise systems use execution trace as input and return a summary of its main content in the form of abstract execution events [8], or as UML sequence diagrams [5]. In the remainder

26th IEEE International Conference on Software Maintenance in Timişoara, Romania
978-1-4244-8628-1/10/$26.00 ©2010

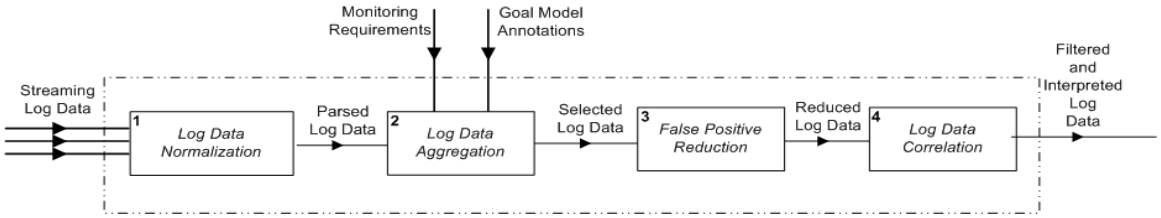

Fig. 1. Logical architecture of the log filtering and interpretation framework.

of this section, we describe related work in the areas of RCA and LSI.

A. Root cause analysis

In the context of software maintenance, RCA (also known as fault localization) represents a class of techniques for the detection and identification of the concealed fault(s) that is (are) at the source of a system failure or a user reported incident. Although most of the literature on RCA has been in the area of low level communication systems; however, the arrival of service oriented architecture has led to more interest in developing and adopting RCA techniques in higher layers such as business processes and software services layers. The approaches in the literature can be classified as based on probabilistic approaches such as Bayesian Belief Networks [13], or based on decision trees [2] or rule sets [6] or decision matrices [9], [10]. Some approaches rely on natively generated low level trace information [2] others generate their own log data using instrumentation [14] or by intercepting system calls [17]. Hanemann proposes a hybrid reasoning approach where the root cause of an incident is identified by first searching a set of rules that map symptoms to root causes [6]. Yuan [17] collects trace data and compares it to already collected sets of trace data. Similarly, Steinder [13] analyzes the symptoms in a system and uses Bayesian Belief Networks to model end-to-end services in the system and compute the Most Probable Explanation set. In [14], Wang et al. instruments the source code of monitored applications and use the generated log data to verify the requirements satisfaction as well as for system diagnostics. Unlike [14] which relies on instrumenting software systems, our approach relies on natively generated log data which is less intrusive and thus a more practical approach when analyzing off-the-shelf commercial products. In addition, our approach handles diversity in log data (syntactically and semantically) unlike other approaches such as [2] where the data format and contents are known a priori. Similarly to [13], our approach includes modeling the system to be monitored before using the RCA framework; however, the preparatory effort is less than for rule based approaches which require building association relationships [6], [9]. Similarly to all other approaches relying on raw log data, the diagnosis produced by framework directly depends on the quality of the collected log data.

B. Latent Semantic Indexing

LSI is an indexing and retrieval method to identify relationships among terms and concepts in an unstructured collection of text. LSI was first patented in 1989 by Deerwester et al. [3]. LSI is commonly used in areas such as web retrieval and document indexing [15] and feature identification [12]. In our work we consider each log entry as a document. In this respect, LSI can be used for identifying those log entries that mostly associate with a particular user query denoting a system feature, a precondition, an occurrence or an effect of an operation. In addition, we have introduced a distance function to estimate the weight for fields where traditional semantic analysis does not apply (such as timestamp).

III. Log Filtering and Interpretation Framework

A. Overall Architecture

As illustrated in Figure 1, the proposed framework has four components. The first component is log data normalization component. It transforms streaming log data into a common format and stores it into a centralized database table. The second component is a log data aggregation component. It uses a set of constraints defined in object constraint language (OCL hereafter) to generate an equivalent set of SQL queries, which in turn are used to extract a subset of log data from the log database. The third component refines the data extracted earlier by reducing the false positives using a LSI based clustering technique. The processed log data are then fed to a correlation component which classifies events according to what transaction or business process instance they belong with. The output of this framework is a collection of highly cohesive log data sets that can be used to verify/deny a goal model by using a RCA algorithm to produce a tractable set of diagnoses showing which components may have failed.

B. Framework Inputs and Output

1) Log Data: The first input to the monitoring framework is the natively generated log data. Log data are obtained as sequences of events from different loggers.

2) Annotated Goal Models: The second input to the framework is a set of annotated goal model(s), each representing the requirements of the monitored applications. Goal models can be developed either manually by the system analysts or through reverse engineering the source code [16] and are built a priori to using the monitoring framework. Figure 2 illustrates an example of a goal model representing the *Apply_For_Loan* business process used in our test scenario. Goal models represent an AND-OR tree decomposition of the functional and non-functional requirements of the systems being monitored. We annotate goal models with *precondition*, *occurrence*, and *effect* predicates that are used by the framework to generate queries and extract a collection of log data that in turn are used to verify the particular properties of the goal model, or equivalently the functional or non-functional system requirement, being considered.

3) Monitoring Focus Qualifiers: The third input is the collection of monitoring *qualifiers* that represent the user's particular points of interest such as time interval, server name,

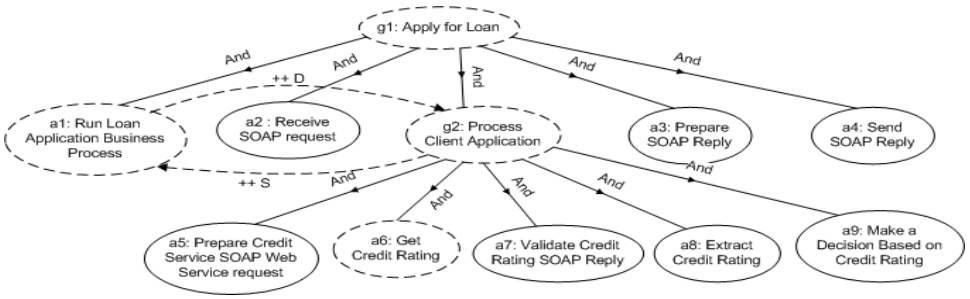

Fig. 2. Goal Model of the Business Process *Apply_For_Loan*

IP address, and user names, etc. The monitoring *qualifiers* are expressed using OCL expressions.

4) Framework Output: After post-processing the results of the SQL queries, the framework produces subsets of interpreted log data as a stream of logical literals. The presence (or lack of) of the logical literals represents the occurrence (or the non-occurrence respectively) of the set of events of interest (i.e. preconditions, occurrence and effects of the goal model nodes). This generated stream of literals is most useful for a recent RCA technology in software systems based on the propositional satisfaction of system properties using SAT solvers [14]. More specifically, given a goal model represented as a propositional formula f, the propositional satisfiability (SAT hereafter) problem consists of finding values for the variables of the propositional formula f that can make f evaluates to true using the stream of logical log data literals as evidence of the satisfaction/denial of individual goals/tasks.

IV. Log Normalization

Log normalization consists of two steps: first, designing an unified log schema, and second, building a set of model transformations. In the first step, we design an unified schema, that we call *LogData*, to contain fields that are common to the different logging systems in our test environment as well as fields that we deem necessary for our analysis. We use as a reference in our schema design the WSDM Event Format (WEF) standard which is an OASIS standardized version of the IBM's common based event (CBE) [7], [11]. We also add to *LogData* some fields that we considered to be essential but did not appear in WEF such the *EventTypeID* field from Windows Event Viewer which represents a class of events and is different from the *EventRecordID*. *LogData* contains the following fields: *EventRecordId*, *ReportTime*, *SourceComponent*, *SourceComponentAddress*, *EventSituation*, *CorrelationId*, *EventTypeId* and *Description*. The second step consists of building model to model transformations to map fields from individual log sources into the unified schema.

V. Log Reduction

As described earlier, log reduction limits the size of the log data that need to be considered for verifying or denying a diagnosis hypothesis generated by the SAT solver.

A. Formation of Goal Annotations

Nodes in the goal models are annotated with OCL expressions pertaining to *pre-conditons*, *occurrence* and, *effect* constraints. Furthermore, each of these OCL goal tree annotations Q_a can be augmented with monitoring *qualifiers* Q_m, that are optional user imposed filters, to yield a more restrictive OCL expression Q_r. Such *qualifiers*, provide specific limit values that relate to the name of the process affected, the name of the server involved and the time interval by which the analysis is applied on. These qualifiers aim to increase the performance of the log reduction process. For space limitations we do not show the details of the augmentation algorithm.

B. Log Reduction based on SQL Query Generation

The first log reduction technique we consider is based on the generation of SQL queries for each augmented OCL expression for a goal tree node and the application of these SQL queries against the log data pool.

An OCL formula generally includes the following components: context, def, inv, pre, body, and post.

A particular type of expression appearing in the *body* of an OCL formula is the *select* OCL body operation that has the general syntax:

$$collection \rightarrow select(logical\_expression)$$

In this context, the OCL logical expressions we have presented as annotations of the goal tree nodes in the previous section (e.g. the OCL augmented expression Q_r), correspond to the *logical_expressions* that will eventually appear in the select body operation of the complete OCL formula for each goal tree node. For space limitations we present here only the *select* logical_expression operation of the OCL formula instead of the complete OCL formula. The rest of the elements of the complete OCL formula for each goal tree node are trivially constructed where the *def*, *pre*, *post* and, *inv* sections are set to NULL, and the *context* section is the name of the goal tree node.

In turn, the $(logical\_expression)$ of the OCL *select* operation is a Boolean formula consisting of $atomic\_logical\_expressions$. Each atomic logical expression is of the form:

$$(self.attribute\_name,\ comparison\_operator,\ attribute\_value)$$

where $attribute\_name$ represents an attribute of the log data, $comparison\_operator$ is of the form $(=, <>, <, >, >=, <=)$, and $attribute\_value$ is the value of $attribute\_name$.

The SQL query process generation is then as follow:

1) Create skeleton for SQL Query. The skeleton is of the form:
   SELECT $*$ FROM <Log DataBase Table> WHERE <generated SQL query expression>
2) Create the <generated SQL query expression> from the logical_expressions in a *select* operation in the body of the corresponding OCL formula for the given goal tree node. The creation of such SQL query expression is a transformation that:
   a) maps each OCL logical_expression of the form:
   $(self.attribute\_name$
   $comparison\_operator$
   $attribute\_value)$
   to
   $attribute\_name$
   $comparison\_operator$
   $attribute\_value$ SQL expression and,
   b) maintains the structure of the Boolean expression in the OCL expression to the corresponding SQL query expression. For example, the OCL expression Q_r:

Q_r :: self.report_time > T1 and self.report_time < T2 and self.source_component = 'soapUI' and self.description.contains('receive loan application')

leads to the corresponding generated SQL expression below:

$SELECT \star\ FROM\ [EVENTS].[LogData]$
$WHERE\ report\_time > T1\ AND\ report\_time < T2$
$AND\ source\_component = 'soapui'$
$AND\ description\ like\ \%receive\ loan\ application\%$

C. Log Reduction based on Latent Semantic Indexing

The second log reduction technique we consider is based on LSI [3]. This technique has the advantage that in order to increase the recall of the process, we allow users to use more relaxed queries and to obtain ranked results as opposed to using SQL queries to obtain one fixed result. LSI is a technique commonly used in web search for ranking and indexing documents (here the log data represent the documents searched and indexed). In a nutshell, the LSI based algorithm can be outlined as follow:

The first step is to create a vocabulary by extracting keywords from the log data. These keywords represent concepts, actions, etc. During this process, all capitalization, punctuation and extraneous markup are stripped away. The second step is to refine the vocabulary above by applying a stop list and removing commonly used words that do not carry semantic meaning. The third step is to apply a stemming process by removing common endings from words, leaving behind an invariant root form. The fourth step is to create the term-document matrix. The rows of the matrix represent a set of signature vectors each corresponding with a log entry. The columns of this matrix are the keywords in the vocabulary generated earlier. The signature vector is populated using a term weighting function that favors keywords that characterize the corresponding log entry, and disfavors keywords that are common to all long entries. The fifth step consists of transformation of the term-document matrix into a lower dimensionality matrix using the SVD algorithm. The transformed matrix is called the concept-document matrix. SVD allows for the factorization of a matrix A into a product of three matrices $(U * S * V^T)$, where S is a diagonal matrix containing the singular values of A. By choosing the highest k singular values, we can achieve a dimension reduction as follow $A_k = U_k * S_k * V_k^T$. Finally, the user query is also considered as a document and is also converted into a vector in the reduced concept-document space. This vector is then multiplied by the set of row vectors representing the individual log entries in the concept-document frequency matrix resulting in another vector that represent the similarity of the query against the corresponding log entries.

TABLE I
SQL AND LSI FALSE POSITIVE AND NEGATIVE COMPARISON TABLE

	Total Rows		False Pos.		False Neg.	
SQL Where Condition	SQL	LSI	SQL	LSI	SQL	LSI
description like "Message Broker started%'	0	199	0	198	1	0
'%MB7Broker%' or description like '%start%'	193	200	192	199	0	0
'%database%' and description like '%start%'	6	22	5	21	0	0

Note that the two log reduction methods described above are presented as alternative approaches. The first approach could be characterized with low false positives and high false negatives. In fact, when we experimentally filtered log data by directly applying the generated SQL queries, we noticed that unless the SQL queries are very relaxed and well formulated, the filtered log data returned an empty set. On the other hand, LSI based log filtering is characterized by higher false positives and lower false negatives. Theoretically, it is possible to use the two approaches in a complementary fashion (after relaxing the SQL queries in the first approach) by applying them sequentially to get better results and lower false positives.

VI. SOA/BPM EXPERIMENTATION ENVIRONMENT

The test environment that we use to illustrate the proposed framework contains a set of off-the-shelf applications and emulates an enterprise environment. The experimentation environment includes a business process layer and a service oriented infrastructure, and is built using commercial off-the-shelf software such as IBM WebSphere Business Process, IBM WebSphere Message Broker and Microsoft SQL Server 2008 database management system.

VII. THE FRAMEWORK IN ACTION

Before using the framework, enterprise systems are modeled using annotated goal models and then executed with event logging enabled. The framework collects diverse logs, generates queries that would filter and integrate the log data, which in turn provide useful information for RCA. To evaluate the precision and recall of filtering log data using the proposed

framework, we use the test environment described in the previous section to run a set of experiments and we measure the false positive and false negative results with respect to finding log data relevant to a given query.

Our test scenario is based on a custom built financial business process *(Apply_For_Loan)* deployed on the IBM process server. The test scenario involves an online user applying for a loan and having their loan evaluated and finally a loan accept/reject decision is made based on the information supplied by the user. We use the two information retrieval techniques: SQL query and LSI the log data incoming from five different monitoring sources. The log database table contained 501 log entries transformed from their original format and stored in the unified format. The filtering results are shown in Table I. Queries used in this experiment are of the form *SELECT * FROM [EVENTSDB].[dbo].[LogData] where description like "%Message Broker started%"*, but for space limitation, we only show WHERE condition in Table I. After performing LSI, the extracted vocabulary contained 570 keywords. The rank reduction was done using a k value of 50 corresponding to a minimum of 3 for singular values. Documents with a threshold value of $\alpha > 0.5$ were considered as relevant to the evaluated queries. We note that for higher values of k, more concepts are generated, resulting in less false positive but potentially more false negatives in the filtered log data.

VIII. CONCLUSIONS

This paper presents a framework for log reduction and interpretation in distributed systems. In this paper, we focus only on the normalization and the aggregation/selection phases. We inspect log data generated by six off-the-shelf software applications. Based on the above experimentation, we define an unified log data format that fields from both standards (WEF and the Windows Event Viewer) deemed necessary. In terms of log data reduction, we enhance the basic SQL queries recall by using the LSI which is an approach commonly used in web search algorithms. This enhancement leads to less false positives and easier formulation for queries.

Our main motivation to use goal models is to make the log data extracted by our framework readily available to be used by SAT based RCA tools for proving/disproving their diagnosis. The filtering techniques and algorithms in the proposed framework can be applied to different types of requirement models (e.g. UML models) or runtime models (e.g. BPEL), as well as annotations based on predicate logic or JML (Java Modeling Language). In fact, by interpreting the filtered data according to the proper context, the proposed framework reduces log data for other purposes such as intrusion detection, monitoring or understanding of software systems.

The quality of the reduced log data depends on the quality and availability of the original log data. Low quality log data such as log data with too few fields or with generic description can lead to a degradation of the framework's precision/recall. On the other hand, the availability of the log data showing the internal sequence of events in a system is essential in order to diagnose correctly the system. For example, one of our experiments involved the injection of some faults, and then the application of our framework on the resulting log data. Although the failure in the application did occur as expected; however, this failure propagated in some instances to the corresponding event/log generation subsystem causing it to fail as well, which potentially lead to reduced accuracy in our interpreted log data.

Finally, we would like to mention that this work is supported by a grant from CA Labs.

REFERENCES

[1] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202, New York, NY, USA, 2004. ACM.

[2] M. Chen, A. X. Zheng, J. Lloyd, M. I. Jordan, and E. Brewe. Failure diagnosis using decision trees. *Autonomic Computing, International Conference on*, 0:36–43, 2004.

[3] S. Deerwester, S. T. Dumais, G. Furnas, L. T. K., and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41:391–407, 1990.

[4] P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani. Reasoning with goal models. pages 167–181. Springer, 2002.

[5] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *ICPC*, pages 181–190, 2006.

[6] A. Hanemann. A hybrid rule-based/case-based reasoning approach for service fault diagnosis. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 734–740, Washington, DC, USA, 2006. IEEE Computer Society.

[7] IBM. Common base event. http://www.ibm.com/developerworks/library/-specification/ws-cbe/, 2009.

[8] Z. M. Jiang, A. E. Hassan, P. Flora, and G. Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *QSIC '08: Proceedings of the 2008 The Eighth International Conference on Quality Software*, pages 181–186, Washington, DC, USA, 2008. IEEE Computer Society.

[9] S. Nadi, R. Holt, I. Davis, and S. Mankovskii. Draca: Decision support for root cause analysis and change impact analysis for cmdbs. University of Waterloo, CA Labs, Canada, 2009.

[10] M. Natu and A. Sethi. Using temporal correlation for fault localization in dynamically changing networks. *International Journal of Network Management*, 18:4, 2008.

[11] OASIS. Wsdm event format. http://docs.oasis-open.org/wsdm/wsdm-muws1-1.1-spec-os-01.pdf, 2007.

[12] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol. Combining probabilistic ranking and latent semantic indexing for feature identification. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 137–148, Washington, DC, USA, 2006. IEEE Computer Society.

[13] M. Steinder and A. S. Sethi. Probabilistic fault diagnosis in communication systems through incremental hypothesis updating. *Comput. Netw.*, 45(4):537–562, 2004.

[14] Y. Wang, S. A. Mcilraith, Y. Yu, and J. Mylopoulos. Monitoring and diagnosing software requirements. *Automated Software Engg.*, 16(1):3–35, 2009.

[15] L. Wu, J. Feng, and Y. Luo. A personalized intelligent web retrieval system based on the knowledge-base concept and latent semantic indexing model. *Software Engineering Research, Management and Applications, ACIS International Conference on*, 0:45–50, 2009.

[16] Y. Yu, Y. Wang, J. Mylopoulos, S. Liaskos, A. Lapouchnian, and J. C. Leite. Reverse engineering goal models from legacy code. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 363–372, Washington, DC, USA, 2005. IEEE Computer Society.

[17] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 375–388, New York, NY, USA, 2006. ACM.