A Language for Describing Disruptive Application Event Patterns Based on Combined Log Sequence and Concurrency

Denzel D. Gabriel[1], Ralph Laurence M. Matienzo[1], Aresh T. Saharkhiz[1] and Zaliman Sauli[2]
[1]*School of Information Technology, Mapúa University ,Makati 333 Sen. Gil Puyat Ave., Makati City 1200, Philippines.*
[2]*School of Microelectronic Engineering, Universiti Malaysia Perlis, Pauh Putra Campus, 02600 Arau, Perlis, Malaysia.*
ddgrabiel@mymail.mapua.edu.ph

***Abstract*—Disruptive events are usually preceded by some signs or states that can usually detect by a running application and recorded in the application log; these signs are typically recorded as warning level log entry. Warning logs do not by themselves lead to disruptive incidents, but it is indicative of something that can go wrong. Combined with the other log entries it can be conclusive of an impending incident or disruptions. This study determines the language that will best describe these scenarios together with their resolution accurately; The language can be understood by humans and interpreted by a computer program can raise the alarm and sent notifications; allow either manual or automated resolution to be applied. The goal is to pre-empt the actual occurrence of a costly incident and the speedy application of corrective measures. Results showed that the language is able to successfully describe all the possible log scenarios of interest occurring in sequence, parallel or combined.**

Index Terms*—Application Logs; Language; Log Monitoring; Mathematical Induction; Pattern Recognition; Programming Language.

I. INTRODUCTION

A programming language is a low or high, machine level language used to write instructions and construct computer programs such as C++, C#, COBOL, SQL, Java, Assembly Language and many more [1]. The Authors can classify them into one or a combination of the most common programming paradigms such as imperative, object-oriented, declarative, or functional. Imperative language is a programming paradigm that uses a sequence of statements to determine how to reach a certain goal. The basis of object-oriented programming is the concept of objects that contain properties and behaviors. Declarative language is a very high-level programming language where that programmer needs to specify only what to do rather than how to do it [1-4]. An example of this is SQL or Server Query Language used in the database. You just need to select rows or columns in the database, and it is up to the programmer / user what to do with those selected areas. Lastly, a Functional programming language is a programming paradigm where you set a variable or string, and make some functionality of it. The more appropriate programming paradigm to use for the design of a language depends on its purpose and intended implementation.

In every language constructed, there is syntax and semantics. Each language has a different syntax and semantics based on its constructed rules. Syntax defines the arrangement and combination of various symbols of the language. Semantics is the meaning of the language constructed.

Application Management and Support (AMS) is a growing multi-million dollar business whose objective is to ensure smooth and continuous business operation. The Service Level Agreement (SLA) between suppliers and the customers formally and quantitatively defines this objective.

Application maintenance relies heavily on reacting instead of predicting events. Commercial applications and services normally come with warning logs that can be useful in predicting potential disaster. The ability to monitor collectively the logs generated by the various components of the applications together will enable management to avoid breaches in service level agreements.

There are existing log monitoring applications that can detect and perform the necessary actions in the event of an incident such as automated notifications and in some cases automated remediation. However, so far the common use of logs is to aid the application management in manually investigating the root cause of the incident in order to determine the necessary fixes or workaround. However, this method of reacting to the events as it occurs causes avoidable delays that can lead to SLA breaches and penalties. Current log monitoring systems are mostly reactive and designed merely to detect and trigger actions after the occurrence of an incident [5-11].

This study focuses on the design of a declarative language. The proposed language will serve as a means to improve the capabilities of existing Log Monitoring Tools. Specifically, the use of the language is to describe the combination of a set of patterns or sequences and concurrencies of the logs of an application to prevent disruptions in running applications. The simplicity and ease of use of the Declarative language make it more appropriate for use for the purpose of this study. Even non-programmers can use the proposed language. This study will explore a proactive approach in avoiding rather than fixing issues that can occur. The proactive approach is to find combined patterns of events occurring sequentially and or in parallel that can lead to disruptive situations.

II. MATERIAL AND METHODS

The authors will provide the proposed language to enable the user to describe the combination of concurrent and sequential log patterns. In this study, the authors will investigate the logs generated by Alfresco, which is a type of Enterprise Content Management (ECM) System application. The methodology of this study comprises of several parts,

such as identify and classify log formats, identify permutations of Log events, establish a suitable language syntax and semantics, develop a prototype, and language validation.

A. Identify and Classify Log Formats

The authors identified and classified different applications based on the purpose. Different applications have slightly different log format. The next step is to identify standard log formats used in each of the application classes. This includes logs related to input-output events, logs related to user events, transaction-related events, and resource-related events and many more. The final step will be the identification of the essential parts of the log format such as date and time stamp, a location such as a server name, the cause of the event, a detailed description of the event, log level, and others. The logs were grouped based on the following: (1) One string only, (2) Two string with one (1) variable, and three (3) Three or more strings with two (2) or more variables. Logs are just strings separated by varying values.

B. Identify Permutations of Log Events

Identify different combinations of how the different log formats, leading to disruptive situations, can occur. Log events can occur in sequence and in parallel. Figure 1 shows example of log scenario describing sequence of logs. The Authors, used permutation to identify all possible log combinations that can describe a scenario. The log contains the following: timestamp, log level and log details. **Timestamp** is the exact time that the log event occurred. This is the basis for filtering related log events. **Log Level** is the severity or significance of the log. Possible values can ERROR, WARN, INFO or DEBUG. In this study, only the WARN logs will be considered. The WARN are usually generated to indicate some suspicious events that is worth looking into. **Log Detail** distinguishes one log from the other. The same application can generate related logs in sequence. The log sequence will describe the sequence of specific log details and the period between the occurrences of successive log events. If Log1, Log2 and Log3 occurred in sequence, The Authors can describe this as L1 T1 L2 ... T(N-1) LN. The number of possible logs will depend on the possible values of N. If it represent the minimum value to be Nmin and the maximum as Nmax respectively and consider F as the smallest possible fraction value, then to compute for the maximum number of possible values.

If T1 is 20 seconds and T2 is 30 seconds, then this expression says Log Detail 1 is followed by Log Detail 2 after 20 seconds, Log Detail 2 is followed by Log Detail 3 after 30 seconds.

In a realistic case, it may be after some time range. For example, it will be looking for the sequence where L2 follows L1 (a) between 20 to 25 seconds, (b) within 20 seconds, or (c) more than 20 seconds.

The number of possible logs will depend on the possible values of N. If it represents the minimum value to be Nmin and the maximum as Nmax respectively and consider F as the smallest possible fraction value, then to compute for the maximum number of possible values, it used the following:

$$P = 1 + (N_{max} - N_{min})/F \quad (1)$$

If the value is 3 to 6 and the smallest fraction is, 0.5 then this will give them:

$$P = 1 + (6 - 3)/\,0.5 \quad (2)$$

$$P = 7 \quad (3)$$

Giving them 3, 3.5, 4, 4.5, 5, 5.5, 6. Usually, it shall only be interested in a subset of the possible values. This is normally stated as a range of values such as values greater than X, less than X, between X and Y, outside of X and Y or a set e.g. {3,4,9}. The proposed language needs to represent these cases.

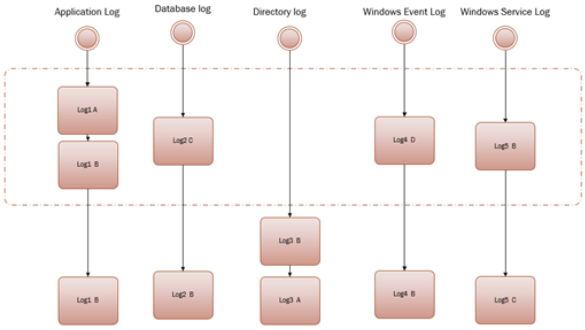

Figure 1: Example of log scenario describing the sequence of logs

C. Establish a Suitable Language Syntax and Semantics

The language syntax should be able to represent all kinds of log formats as identified in the previous step [12]. This syntax should also describe time-based characteristics such as sequencing and parallel occurrence of events. The Authors developed a formal description of the syntax with the corresponding semantics or description of what it means.

The syntax should describe how to declare respondents, actions to perform, and the different log events. In addition, it should be able to describe the sequence and concurrence of log events. This will aid the potential user in describing combined log events to monitor. For a given example of LogA - {10+20} LogB +{5} LogC: N: I.

This can be read as if LogA is followed by LogB within 10 to 20 seconds, and LogC occurs within five (5) seconds after the occurrence of LogA, then N (notify a person) and I (give instruction). The plus (+) and minus (-) signs in the above example represent the occurrences in parallel and in sequence, respectively.

The authors used the following notations to represent and evaluate the Log Sequence and Concurrency:

1. Binary Concurrency (BC) – involves only two parallel logs.
   L1 + {To,Tf} L2
2. General Concurrency (GC) – involves more than two parallel logs.
   L1 + {To1,Tf1} L2+ {To2 , Tf2} L3... (LN – 1) - {ToN , TfN} LN
3. Binary Sequence (BS) – involves only 2 sequential logs.
   L1 - {To,Tf} L2
4. General Sequence (GS) – involves more than 2 sequential logs.
   L1 - {To1,Tf1} L2 - {To2, Tf2} L3 ... (LN – 1)- {ToN, TfN} LN
5. Combined Sequence and Concurrency – involves a combination of parallel and sequential logs.

L1 + {To1, Tf1} L2 - {To2 , Tf2} L3...

In the above notations, {To1, Tf1} represents the time range. Logs are concurrent if it occurs in different applications or servers. It is sequential if it occurs in the same application or server.

The notations given above will serve as the basis for the syntax of the proposed language. Moreover, the proposed language should be able to eliminate semantic ambiguity.

D. Develop the Prototype

The Authors developed an implementation prototype of the log-monitoring program to demonstrate the proposed language. This prototype will make use of dummy applications generating the different log formats. A simple test log generator program simulating the Alfresco application and the service applications will be written to simulate the generation of the concurrent and sequential logs. It used a test log generator to accept a script that describes how to generate the log string and timestamp and how to append it to a log file. The applications that will be simulated to generate logs are the Alfresco Explorer, Alfresco Share, Lucene (Solr) Search Service, Tomcat Web Server and MySQL or PostGress Database event log.

The Authors developed a log monitoring service prototype for tracking the logs. This service will perform the following: (1) monitor log changes, (2) match the log changes against the known log patterns, (3) store the log details in the database, (4) determine if the database matches a configured log scenario, and (5) write the result describing the log pattern in a file if it detects a match.

A prototype compiler was also developed for the language to demonstrate how to test the different scenarios. It will identify a set of scenarios to represent and cover all log combinations based on the knowledge of the parameter domain or range of values derived mathematically as described in the previous sections.

The procedure of the prototype will include a number of scenarios representing each of the following possible combinations: (1) purely single, (2) two sequential or more logs in sequence, (3) two concurrent or more logs in parallel, (4) combined sequential and concurrent, and (5) compound combination. To validate the correctness of the language, it wrote at least two (2) sets of scripts for the possible log combinations. The first set of log scripts will determine if the log monitor properly catches a specific log scenario. The other set of log scripts will determine if the log monitor ignores a non-matching scenario.

The prototype is based on the Backus Naur Form (BNF) specification. The BNF will have the syntax and semantics of the proposed language. The BNF will be used in the Editor where the user can describe the scenarios. The Editor will only produce the correct output that is the description of the scenario when the user follows the correct syntax based on what is written in the BNF.

E. Language Validation

The essential information needed to describe the log patterns to monitor includes the application definition, respondents, log definition, and scenarios. The log definition is the string or log format making up a specific log used by the log monitor in recognizing and identifying the log. This is simply a string possibly containing some numerical values. The scenario is a combination of sequential and concurrent logs. The proposed language should describe these combinations.

The main effort of this study is on the description of the various log scenarios. It can validate that the language is sufficient by proving that the proposed language can represent all these scenarios.

It can organize all the possible description of logs as a well-ordered set. If it uses the symbols L, I, + and - in the description of a scenario, it can write the following:

Case 1: Sequential log pattern
L1
L1 - I1 L2
L1 - I1 L2 - ... - Ln

Case 2: Concurrent log pattern
L1 + I1 L2
L1 + I1 L2 + ... +In-1 Ln

Case 3: Combined Concurrent and Sequential log pattern
L1 + I1 L2 - I3 L3 - I4 L4 ... In-1 Ln

In this example, L is the set of logs that form the scenario, I, is the interval between sequential events or the interval when two or more logs can be considered concurrent, + indicates 2 or more concurrent and – indicates two or more sequential logs. induction method as stated in [13] was utilized to check the correctness of the language. There are two induction steps. The first step, known as the base case, is to prove the given statement for the first natural number. The second step, known as the inductive step, is to prove that the given statement for any one natural number implies the given statement for the next natural number.

F. Test Procedure

The procedure contains three (3) parts, which are the creation of the scenario using the proposed language, simulating the logs generated by the application, and testing the log monitoring prototype.

III. Results and Discussions

The scenarios that are tested are the logs occurred in single, sequential, concurrent, and combined respectively. The images shown in Figures 2 to 5 are the scenarios of the log events. The application definitions were defined, as well as, the respondents and it will notify them if a log has been detected in a specific timeframe, and the log definitions that contains details of the log. The scenarios were also defined but in the different event to determine which respondent should receive the notifications. If a log is detected within a timeframe, it gives notification to the respondents.

Figure 6 illustrates the SQL scripts output in this study. All the definitions are ready to be stored in the database. If there is a need for modification, it is accessible for changes. The user can now create new scenarios and also update or delete the scenarios given if it is necessary as shown in Figure 7.

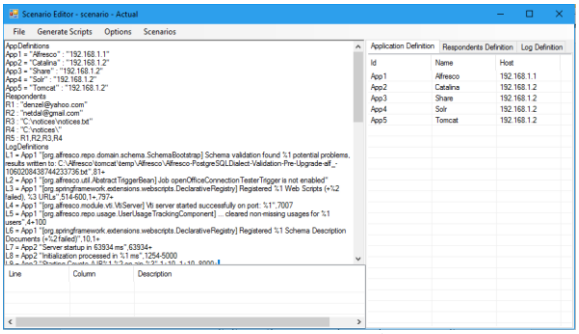

Figure 2: Scenario Editor – Actual (AppDefinitions, Respondents, LogDefinitions)

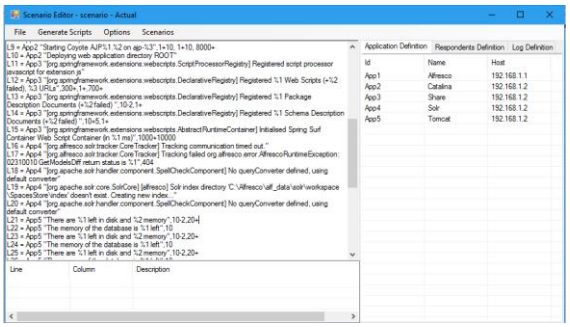

Figure 3: Scenario Editor – Actual (LogDefinitions)

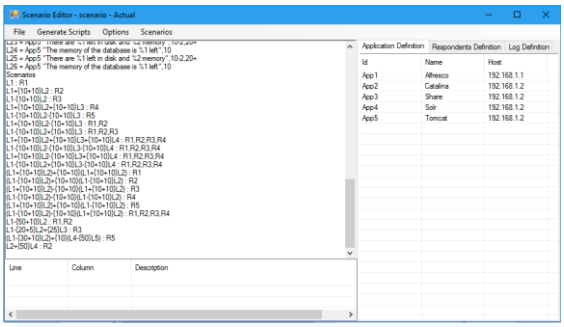

Figure 4: Scenario Editor- (Scenarios)

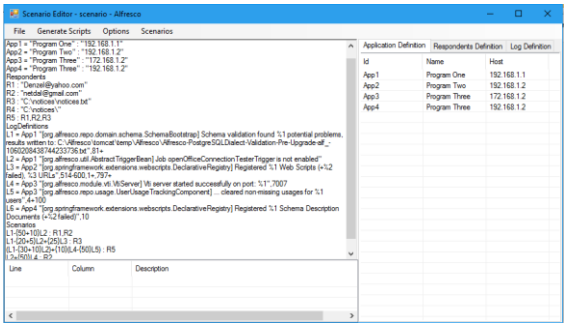

Figure 5: Scenario Editor - Alfresco

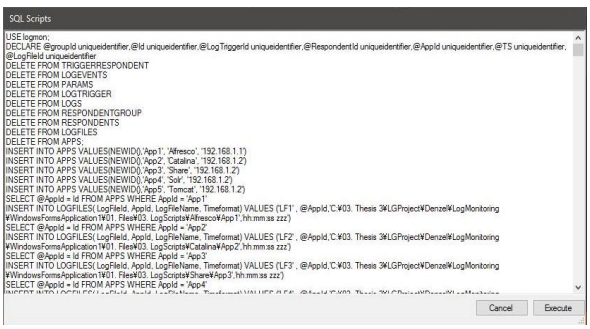

Figure 6: SQL Scripts

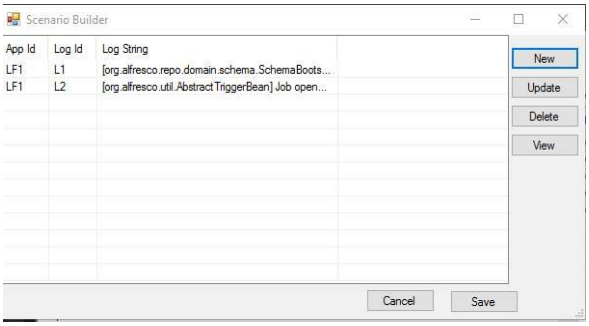

Figure 7: Scenario Builder

The image is shown in Figure 8 an example of the log detection while monitoring, this is real-time detection of the scenario, in a given time interval. The log details are ten (10) left in disk and twenty (20) memory, which is an example of a log event while monitoring the log files. Originally it only detects when there is a newly generated log that occurred.

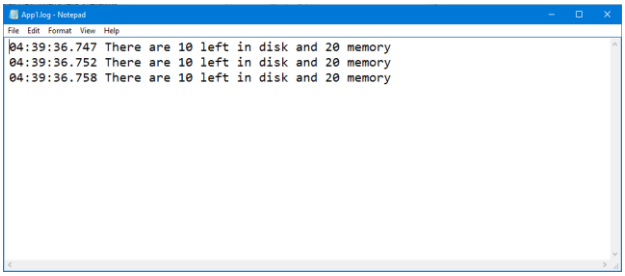

Figure 8: App1.log

Image shown in Figure 9 is detecting a log with a matching scenario. There were some challenges encountered during its development and tested these are found in the Log Format, Parsing, and SQL Code Generation.

Log Format Issue - There are cases where a part of the log can have different values. If it is a number, then this could be specified by range, for example, all positive integers, numbers between minimum and maximum like one to ten. If it is a string, then it can be specified using a wildcard or a set of acceptable values like: {app1, app2, app3}. This may be considered as a limitation.

Parsing Issue - There are tokens that can also be parsed as a string enclosed in a quote. There should be a way to distinguish generic strings from filenames for example. This was resolved by identifying special characteristics of the filename such that it can only begin with a number or letter.

SQL Code Generation - When generating the SQL statements, the presence of a quote character might also show some issues since it is a special character in SQL statement. This is done by preprocessing the string and replacing the single quote with two quotes.

There are probably some issues that are not mentioned and not yet encountered by the Authors; however, these issues will certainly not cause much harm or damage in the operation of the prototype.

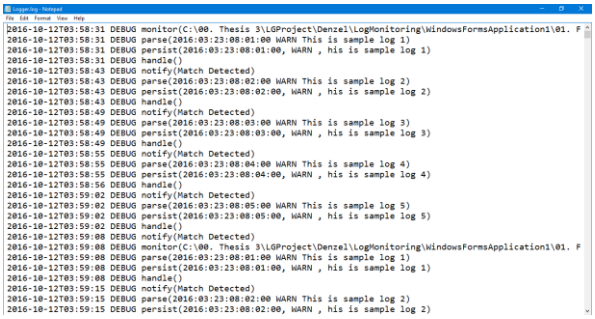

Figure 9: Logger.log

IV. CONCLUSION

The proposed Declarative Language was developed successfully using the methodology defined in the study. Using mathematical approach, The Authors were able to make the classifications and generalizations that became the basis of the development of the rules of the language. The developed Prototype was used to demonstrate how the language will be used in the actual setting. Based on the results and tests made, it can be concluded that the language is able to successfully describe all the possible log scenarios of interest occurring in sequence, parallel or combined. Thus, the developed declarative language was able to represent the classifications and generalizations of scenarios.

REFERENCES

[1] T. L. Hinrichs, D. Rossetti, G. Petronella, V. N. Venkatakrishnan, A. P. Sistla, and L. D. Zuck, "Weblog: A declarative language for secure web development," in *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, 2013, pp. 59–70.

[2] F. Marco and M. Marco, "Design of a declarative language for pervasive systems." Master's thesis, Politecnico di Milano, Milano, 2007.

[3] A. Mottola, "Design and implementation of a declarative programming language in a reactive environment." Università degli Studi di Roma, 2005.

[4] S. J. Løvborg, "Declarative Programming and Natural Language." 2007.

[5] M. O. Shafiq, "Semantically Formalized Logging and Advanced Analytics for Enhanced Monitoring and Management of Large-scale Applications." University of Calgary, 2015.

[6] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson, "SASE: Complex event processing over streams," *arXiv Prepr. cs/0612128*, 2006.

[7] S. Boyer, O. Dain, and R. Cunningham, "Stellar: A fusion system for scenario construction and security risk assessment," in *Information Assurance, 2005. Proceedings. Third IEEE International Workshop on*, 2005, pp. 105–116.

[8] M. Wei, I. Ari, J. Li, and M. Dekhil, "Receptor: Sensing Complex Events in Data Streams for Service-Oriented Architectures.","*Tech. Rep. HPL-2007-176, HP Lab.*, 2007.

[9] W. Fitzgerald, R. J. Firby, A. Phillips, and J. Kairys, "Complex event pattern recognition for long-term system monitoring," in *Workshop on Interaction between Humans and Autonomous Systems over Extended Operation*, 2004.

[10] P. Gay, B. López, and J. Meléndez, "Sequential learning for case-based pattern recognition in complex event domains," in *Proceedings of the 16th UK Workshop on Case-Based Reasoning. 13th December*, 2011, pp. 46–55.

[11] M. M. Burnett and A. L. Ambler, "A declarative approach to event-handling in visual programming languages," in *Visual Languages, 1992. Proceedings., 1992 IEEE Workshop on*, 1992, pp. 34–40.

[12] I. L. Bratchikov, "The Syntax of Programming Languages," *Russ. Nauk.*, 1975.

[13] E. W. Dijkstra, *Selected writings on computing: a personal perspective*. Springer Science & Business Media, 2012.