Automated Log Analysis in Communication Systems

Divjot Kaur, Roshni Chatterjee, Rajdeep Kaur and Surendra Singh
Samsung Semiconductor India R&D
Samsung Electronics
Bangalore, India

***Abstract*— Modern smartphones, servers, routers and other products have complex components, which interact with each other using different protocols. More than a billion such products are sold each year- so it is a herculean task for the different companies involved in the development and sales of such products to ensure that the products work flawlessly without any anomalies. Therefore, any new model of such products must undergo exhaustive testing globally before it can be launched. These wide range of tests generate a huge amount of logs.**

Analyzing these logs manually is a mammoth task. It takes a lot of man-hours across different test sites. This affects the turn-around time required to identify and fix issues. To solve these problems, Automatic Log Analysis needs to be implemented to reduce the burden of analyzing the logs manually, identifying issue logs efficiently, and assigning the issues to right developers, so that developers can focus on fixing the issues rather than having to go through thousands of logs, which may or not have any relevant issue scenarios.

With the recent advancement of supervised machine learning methods, training the computer to perform offline and online log analysis is possible within this industry.

Keywords*— *Log analytics, Machine Learning, Decision Trees, communication systems

I. INTRODUCTION

A bug-free product is essential for the success of any company. Due to various business factors- a product has to be launched in the quickest possible time with the latest technologies adding incremental and sometimes radical changes to existing products or make new modules within products- and these changes invariantly cause bugs. Extensive testing is a critical part of understanding system status and performance issues. During testing, system logs are collected to record system states and significant events at various critical points to facilitate debugging of failures and perform root cause analysis. The following steps are generally performed after log collection is completed:

1. A ticket is raised with the detailed description of the problem and relevant logs.
2. The ticket is assigned to a First-Level-Analysis engineer- who is usually not located at the test site.
3. The logs are then downloaded by the engineer and the issue is analyzed by searching for some specific Keywords / Traces for the failures that can happen.
4. Based on the above analysis, the issue can be declared as normal or abnormal behaviour. Based on the module in which this issue might be occurring, the issue can be re-assigned. Step 3 will be performed again. This process is repeated till the relevant engineer, who is the owner of Module where the issue happened, is finally assigned the issue.
5. The issue is fixed by the module engineer and further testing is done and more logs are analyzed till the issue is found to be fully fixed.

Performing the above steps for a large scale system is time-consuming and requires expert knowledge. Many unknown problems occur in an ever-growing and continuously updated system. Very few engineers want to spend their valuable time analyzing logs. Even the engineer who finally fixes the issue has to repeatedly analyze the logs for several days till the issue is finally declared to be fixed as per the process. Also manual log analysis is error-prone and non-scalable. It also causes stress to the engineers as repeated work eventually becomes boring and this leads to loss of passion for the assigned work. The inefficiency and difficulties of manual log analysis make automated log analysis desirable.

In this paper, we propose to apply machine learning techniques to do automated log analysis as they are effective and efficient to big data problems. Features from the contents of the logs are extracted and learning algorithms are leveraged to detect abnormalities.

The Automatic Log Analysis Tool proposed in this paper does the following:

1. Whenever a new issue is raised or existing issue is updated, a new entry in Action List will be created by the tool.
2. Relevant logs and required binaries are automatically downloaded to decode the logs.
3. The logs are cleaned and relevant features are extracted. This dataset is then fed into our learning model for analysis (this step will be explained in next section)
4. The issue is updated with the analysis. The issue is also classified as Expected or Unexpected behavior. The analysis is emailed to relevant engineer (users).

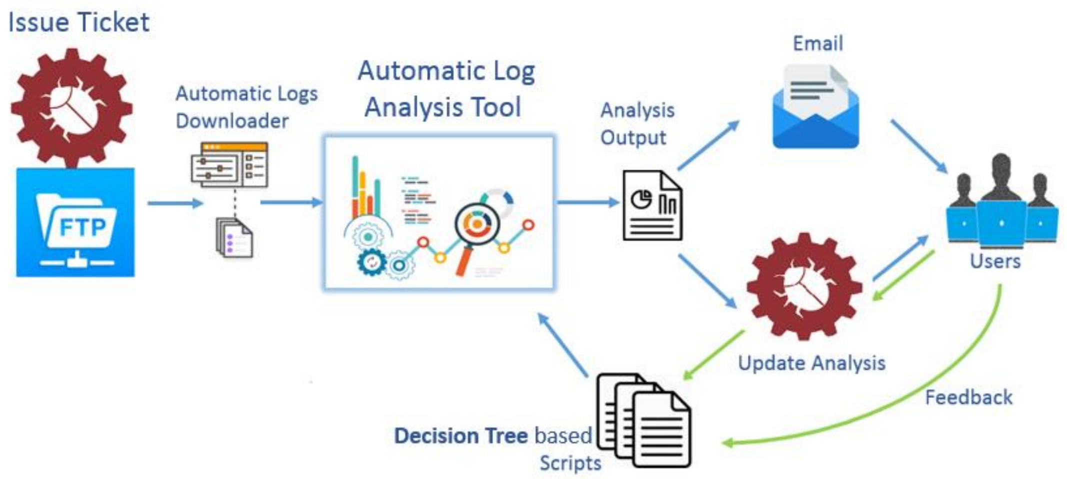

Issue Ticket
FTP
Automatic Logs Downloader
Automatic Log Analysis Tool
Analysis Output
Email
Users
Update Analysis
Feedback
Decision Tree based Scripts

Fig. 1a: Usage of Automatic Analysis Tool in the Ecosystem

Based on the feedback from the engineers and further discussions which are updated on the Issue ticket- the scripts are enhanced to refine the learning model. This is illustrated in Fig. 1a.

II. IMPLEMENTATION

A. Pruning Log by Chosen Events

The unstructured log entries are parsed into structured representation, so that the model can be trained over this structured data (as seen in fig. 1b). Usually a log contains millions of traces, but some traces indicate critical events like :

i. Assert Events, for example, “memory allocation failure”

ii. Failure Events- for example, “call drop notification to user with cause xyz”

iii. Success Events- for example, “Normal Call Release notification to user with cause *PQR*”

These events are added in logs explicitly by developers to identify a particular kind of failure or success case. The “Scripts” in the Figure 1a contain a “Trace List” of such chosen traces only & their usual and unusual patterns.

These events are added initially as part of Supervised Machine Learning process by running the tool on Training Logs. Training logs can be all Conformance Test (like PCT/GCF/RCT/RRM) success logs, Pass logs from issue tickets in Issue tracking system, and other logs that the developer feeds to the system. Later during unsupervised mimicking phase, the tool can add more such events in the “Trace List” by polling the different issue tickets in the Issue tracking system. Logs can be plain text or proprietary logs which are encoded in a specific way. To decode proprietary logs, special binaries and keys maybe needed. For example, “Device IPv4 address is 1.2.3.4 ”is a trace & it is usually present in code as:

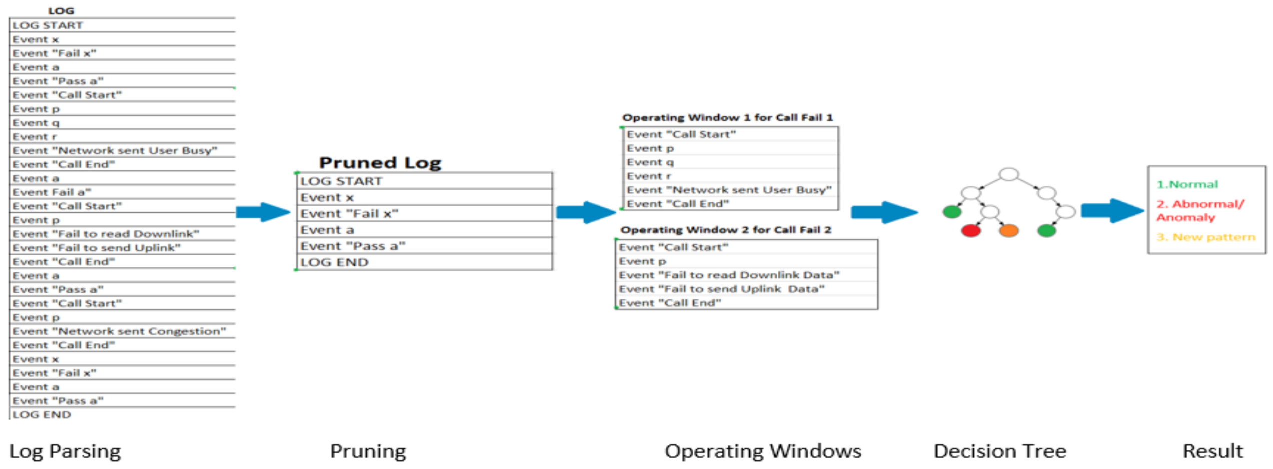
- The constant trace part “Device IPv4 address is %d. %d. %d. %d”
- The variable part with the four “%d” integer values.

The logs can be pruned by choosing only constant traces from logs, which are present in the “Trace List” of Decision Tree based Scripts. This can be seen in fig. 2. This significantly reduces the number of traces and events in final log. Thus, parsing the log becomes much faster.

B. Setting Operating Windows in the Log

Once the log is parsed and pruned to contain only critical events, different Operating Windows are formed based on the critical events in the issue logs. For example, for a Voice Call Drop issue, any event that occurs after the call drop is not important from the analysis perspective. The call drop event is the “End of Operating Window”. Similarly, the start of the call can be considered as the “Start of Operating Window”.

By setting a smaller and relevant “Operating Window”, the amount of computation required decreases significantly. Events like bad signaling conditions, Network Operator messages like “User busy”, etc. are analyzed only during the failure period. If there are multiple Call Drops in the log,



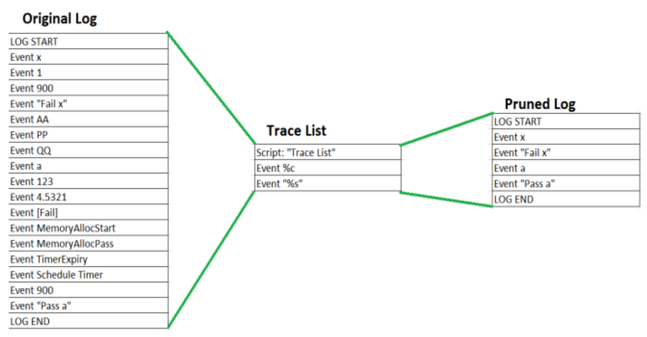
LOG
```
LOG START
Event x
Event "Fail x"
Event a
Event "Pass a"
Event "Call Start"
Event p
Event q
Event r
Event "Network sent User Busy"
Event "Call End"
Event a
Event Fail a"
Event "Call Start"
Event p
Event "Fail to read Downlink"
Event "Fail to send Uplink"
Event "Call End"
Event a
Event "Pass a"
Event "Call Start"
Event p
Event "Network sent Congestion"
Event "Call End"
Event x
Event "Fail x"
Event a
Event "Pass a"
LOG END
```

Pruned Log
```
LOG START
Event x
Event "Fail x"
Event a
Event "Pass a"
LOG END
```

Operating Window 1 for Call Fail 1
```
Event "Call Start"
Event p
Event q
Event r
Event "Network sent User Busy"
Event "Call End"
```

Operating Window 2 for Call Fail 2
```
Event "Call Start"
Event p
Event "Fail to read Downlink Data"
Event "Fail to send Uplink  Data"
Event "Call End"
```

Fig. 1b: Blocks in Automatic Analysis Tool



```
LOG START
Event x
Event 1
Event 900
Event "Fail x"
Event AA
Event PP
Event QQ
Event a
Event 123
Event 4.5321
Event [Fail]
Event MemoryAllocStart
Event MemoryAllocPass
Event TimerExpiry
Event Schedule Timer
Event 900
Event "Pass a"
LOG END
```

```
Script: "Trace List"
Event %c
Event "%s"
```

```
LOG START
Event x
Event "Fail x"
Event a
Event "Pass a"
LOG END
```

Fig. 2: Pruning Log by the trace List formed during training phase

then after analyzing current call drop, the operating window is changed to point of the next call drop in the log.

In Figure 3 example, in “Operating Window 1”, Event “Network sent User Busy” will be found and other Failure events are not found as they are not part of current “Operating Window”.

C. *Decision Tree based Analysis*

Decision Trees are created during Supervised Machine Learning process using traces from training logs from Conformance Testing (like PCT/GCF/RCT/RRM) and logs uploaded as issue tickets in Issue tracking system that have already been analysed by the engineers. The issue tickets usually contain a template with logs and detailed analysis from engineers with log snippets. The tree is created using Python based on the root cause analysis made for the issue observed. The training set is classified into Normal and Abnormal behavior based on the resolution of the issues. The components in the decision tree framework are made reusable and modular so it can be interchanged between the other decision tree algorithms.

During execution within the “Operating Window” the “root node” will check if any of the events mentioned in the “child node” has happened. If the child node has more child events to check, then more scripts can be called like subroutines or a function using the same “Operating Window” or a reduced “Operating Window”. This process is repeated till a classification can be made into:

i. *Normal:* This issue requires no further debugging. It can be a temporary network / environment issue and it is not a device issue. Such issues will be resolved by the tool automatically.

ii. *Abnormal/Anomaly:* This could be a device issue. Based on the events seen in the operating window, the issue is assigned to relevant developer / leader alongwith the analysis.

iii. *New Pattern:* None of the nodes could capture the reason for failure present in the root node / child node. This is a new scenario that has not been captured in the existing script. The tool will assign the issue to a “First Level Analysis” engineer, add it to the Action List and poll the issue in the issue ticket platform. Once the issue is resolved / updated, the new pattern explained by the engineer in the comment section of this issue will be used to train the existing learning model so that such issues can be detected by the tool in the future and be assigned or resolved accordingly.

A basic sample flow is shown in Figure 4. Overtime- each node may have more than 20 branches. And each node maybe called by different scripts. For example: “Downlink Layer 3” script maybe called by “Call End” script and also “SMS end” and “Internet End” scripts- so this allows re-use of scripts in different scenarios and reducing the depth of the Decision Trees.

Apart from detecting different Events- the scripts can also extract values from the events. For example:

Trace: *“Device IPv4 address is 1.2.3.4”* is at timestamp T1
Trace: *“Device IPv4 address is 11.22.33.44”* is at timestamp T2

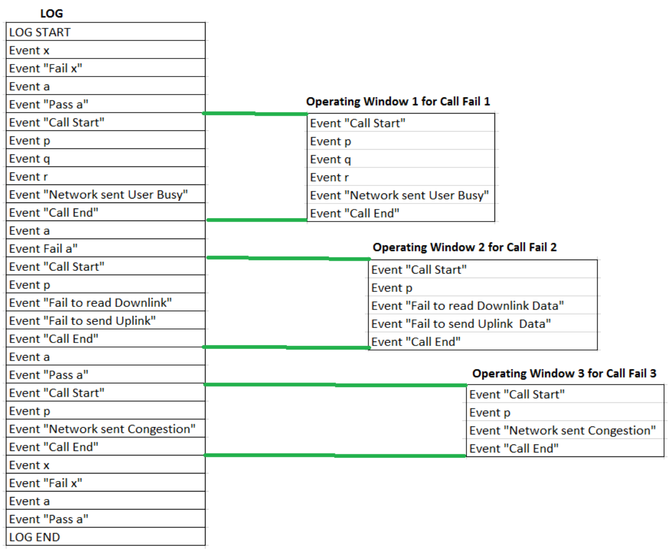
The script can then extract both these values from different timestamps and perform mathematical operations. Then as per the Decision Trees in script, the tool will apply learnt scenario vectors, compare with new input vectors, and generate similar outputs of “Normal/Abnormal/New Pattern” as discussed above. For example, Signal Strength and Signal Quality vectors of Serving Cell in the Modem are going down- then from Training Logs, the tool learns that Neighbor Cell measurement events must be triggered in 2G, 3G, 4G, 5G, CDMA, Wi-Fi etc. and then upon further degradation a reselection or a Handover to be performed to better neighbor cells. During log analysis mode, if the tool detects deteriorating signal strength and quality- it will expect these events. And if such events don’t happen- then it is declared as Abnormal and further investigation from engineer is requested.

III. Evaluation

The tool was run on all the issues found in a certain issue logging system for over three months. Table 1 summarizes the average weekly statistics.

TABLE. 1. Issue Analysis Results

Time Span	**Raw Logs**	**Pruned Logs**	**Total issues**	**Total Issues Auto Analyzed**	**Time taken to auto analyze**
7 days	18,000 Gb	360 Gb	200	180	~40 hours


LOG
Operating Window 1 for Call Fail 1
Operating Window 2 for Call Fail 2
Operating Window 3 for Call Fail 3

```
LOG START
Event x
Event "Fail x"
Event a
Event "Pass a"
Event "Call Start"
Event p
Event q
Event r
Event "Network sent User Busy"
Event "Call End"
Event a
Event Fail a"
Event "Call Start"
Event p
Event "Fail to read Downlink"
Event "Fail to send Uplink"
Event "Call End"
Event a
Event "Pass a"
Event "Call Start"
Event p
Event "Network sent Congestion"
Event "Call End"
Event x
Event "Fail x"
Event a
Event "Pass a"
LOG END
```

```
Event "Call Start"
Event p
Event q
Event r
Event "Network sent User Busy"
Event "Call End"
```

```
Event "Call Start"
Event p
Event "Fail to read Downlink Data"
Event "Fail to send Uplink  Data"
Event "Call End"
```

```
Event "Call Start"
Event p
Event "Network sent Congestion"
Event "Call End"
```

Fig. 3: Operating Windows with different failures

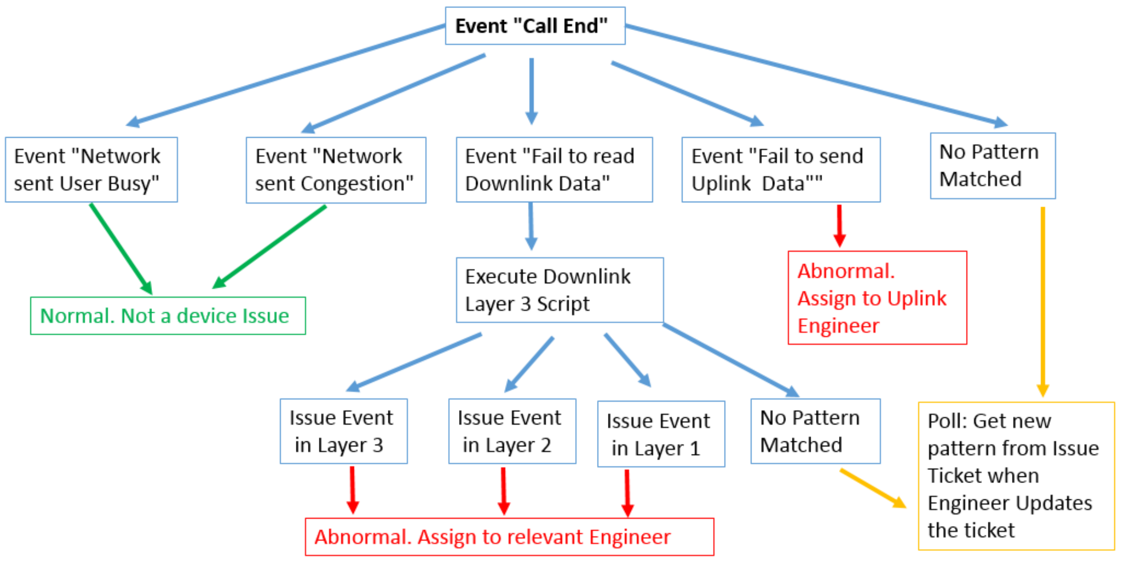

Event "Call End"
Event "Network sent User Busy"
Event "Network sent Congestion"
Event "Fail to read Downlink Data"
Event "Fail to send Uplink Data""
No Pattern Matched
Normal. Not a device Issue
Execute Downlink Layer 3 Script
Abnormal. Assign to Uplink Engineer
Issue Event in Layer 3
Issue Event in Layer 2
Issue Event in Layer 1
No Pattern Matched
Poll: Get new pattern from Issue Ticket when Engineer Updates the ticket
Abnormal. Assign to relevant Engineer

Fig. 4: Decision tree example

It took about 40 hours to auto analyze the logs from 7 days without a single engineer intervention (either Resolved as "No Issue" or "Assigned to right engineers"). To do the same job manually, about 800 man-hours are needed. The burden of 20 engineers is also reduced by avoiding the repeated job of analyzing the logs. 2% false positives were reported; when an engineer was assigned an issue which was actually normal behavior. This was due to the inadequate priority setting during the decision tree formation. The other 8% couldn't be analyzed due to failure to get necessary logs / binaries, as it was missing in the issue ticket. Few of the failures were due to "New Pattern" scenarios for tool. Such "New Pattern Scenarios" were added to "Pending Action List" to add them into the training set. The correctness of the model was evaluated by running it on a large of set of issue tickets during regression. This automated the process of enhancing and adding new patterns into the training database. It was also found that certain tweaks in the algorithm led to better results. For events like Call Fails, the Operating Window where the "Start of Operating Window" & "End of Operating Window" were set using Events like "Start of Call" and "End of Call", the analysis accuracy was higher and execution time was better than other methods of setting the Operating Window. For events like Slow Internet, a sliding Operating Window was more efficient than event-based Operating Window. The Operating Window was made flexible by choosing it based on certain events, states, timer or sliding, based on the event being analyzed.

IV. Conclusion

This paper presents an approach for automated log analysis using decision trees. Compared to other available Automatic Log analysis tools, a new contribution is made by extracting the analysis provided by the module engineer from the Tickets on the Issue tracking platform incorporating in the training set for the learning model based on Decision Trees to adapt to missed scenarios. From the existing literature, it was concluded that unsupervised learning is slow and less accurate than supervised learning [1]. The main advantage of unsupervised learning is that it works for untrained scenarios as well. To overcome this, feedback from Issue ticket logs along with the analysis was incorporated in the training set.

Using this proposed method, the time taken to perform "First Level Analysis" can be reduced by 95% as compared to that taken in manual log analysis. This hastens the process of assigning the issues to the appropriate engineers. The Turn-Around time of fixing issues is greatly improved, which makes the "Go-To-Market" process faster. This also reduces the stress levels amongst engineers performing the "First Level Analysis" of the issue logs. The engineers can focus their time and energy on critical issues and other developmental activities. The Automatic analysis tool has been built by using concepts of Log pruning, Operating Windows and the Supervised Learning model which incorporates feedback from Issue Tickets is able to generate reliable and fast first level analysis of issues. It is certainly worth exploring and implementing Automatic Log Analysis for every major product.

References

[1] Shilin He, Jieming Zhu, Pinjia He, and Michael R Lyu. 2016. "Experience Report: System Log Analysis for Anomaly Detection" Proc. International Symposium on Software Reliability Engineering (ISSRE).

[2] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM

[3] Bhaskar N. Patel, Satish G. Prajapati and Dr. Kamaljit I. Lakhtaria "Efficient Classification of Data Using Decision Tree"

[4] https://pdfs.semanticscholar.org/1fe4/7722d5e65829c7e04b19648f39b22384d28c.pdf

[5] Leonardo Mariani, Fabrizio Pastore "Automated Identification of Failure Causes in System Logs" https://ieeexplore.ieee.org/document/4700316/

[6] Logentries: Log management & analysis software made easy (https://www.loggly.com/docs/anomaly-detection).

[7] Loggly: Cloud log management service (https://www.loggly.com/docs/anomaly-detection).