Aufgabenstellung

zur

Abschlussarbeit
im Studiengang Master Automotive Software Engineering

für

Frau Rashmi Gawande
geb. am 21. Februar 1989 in Indien

zum Thema

Evaluation of Automotive Data mining and Pattern Recognition Techniques for Bug Analysis

Betreuer/ Prüfer: Prof. Dr. Wolfram Hardt
Ausgabedatum: 02.09.2015
Abgabedatum: 09.02.2016
Tag der Abgabe: 

Unterschrift: Prof. Dr. F. Hamker
Vorsitzender des Prüfungsausschusses
Evaluation of Automotive Data mining and Pattern Recognition Techniques for Bug Analysis

Master Thesis

for
the fulfillment of the academic degree
M.Sc. in Automotive Software Engineering

Faculty of Computer Science

Department of Computer Engineering

Submitted by: Rashmi Gawande, 21.02.1989

Supervisor: Prof. Dr. W. Hardt
Ferdinand Kastl

January, 2016
In an automotive infotainment system, while analyzing bug reports, developers have to spend significant time on reading log messages and trying to locate anomalous behavior before identifying its root cause. The log messages need to be viewed in a Traceviewer tool to read in a human readable form and have to be extracted to text files by applying manual filters in order to further analyze the behavior. There is a need to evaluate machine learning/data mining methods which could potentially assist in error analysis. One such method could be learning patterns for “normal” messages. “Normal” could even mean that they contain keywords like “exception”, “error”, “failed” but are harmless or not relevant to the bug that is currently analyzed. These patterns could then be applied as a filter, leaving behind only truly anomalous messages that are interesting for analysis. A successful application of the filter would reduce the noise, leaving only a few “anomalous” messages. After evaluation of the researched candidate algorithms, two algorithms namely GSP and FP Growth were found useful and thus implemented together in a prototype. The prototype implementation overall includes processes like pre-processing, creation of input, executing algorithms, creation of training set and analysis of new trace logs. Execution of prototype resulted in reducing manual effort thus achieving the objective of this thesis work.

**Keywords:** Data mining, Pattern recognition, Trace log analysis, Text mining
Acknowledgements

First of all, I would like to thank Mr. Albrecht Moeckel (Team Manager) and Mr. Ferdinand Kastl (Expert) for giving me the opportunity to perform Master Thesis with Elektrobit Automotive GmbH.

Working under the constant guidance of my immediate supervisor Mr. Ferdinand Kastl was an amazing experience and I would like to extend my gratitude to him especially for guiding and motivating me constantly throughout my thesis work. He not only helped me in researching different data mining algorithms but also provided right direction to find the needle in a haystack. It was not an easy task to conclude on the performance of the algorithms and their results without his valuable feedback and grading on the output. I would also want to express my gratitude towards other team members of HMI department for providing me the right information and immense support during my early phases.

I would also like to thank Prof. Dr. W. Hardt for accepting and registering my thesis topic and showing trust in me to fulfill their expectations. His suggestions during the concept presentation surely proved an important guidance and kept me motivated throughout the thesis work.

Lastly, completing this thesis work was not possible without the moral support of my family and friends. This was a great learning experience where it required efforts in researching several algorithms and test their feasibility by minutely studying their different requirements and desired outputs, thus developing a good automated prototype for assisting in error analysis.
# Contents

1 Introduction ........................................... 1
   1.1 Motivation ........................................ 1
   1.2 Problem Statement ................................ 3
   1.3 Structure of thesis .............................. 3

2 State of the art ...................................... 4
   2.1 Motivation ........................................ 4
   2.2 Existing alternative solutions .................... 6
   2.3 Literature research ............................... 6
      2.3.1 Sentiment analysis .......................... 7
      2.3.2 Principal Component Analysis .............. 9
      2.3.3 Data Clustering ............................. 11
      2.3.4 Apriori Algorithm .......................... 12
   2.4 Evaluation of algorithms ......................... 14

3 Concept .............................................. 16
   3.1 Approach .......................................... 17
   3.2 Pre-Processing Techniques ........................ 20
      3.2.1 Noise reduction ............................. 20
      3.2.2 Filtering .................................... 21
      3.2.3 Sampling .................................... 22
   3.3 Message library .................................. 23
      3.3.1 Regular expressions ......................... 23
      3.3.2 Data optimization .......................... 25
   3.4 Indexing and Structuring ......................... 27
   3.5 Feeding input .................................... 27
   3.6 Pattern recognition ................................ 30
      3.6.1 Rapid Miner tool ........................... 30
      3.6.2 Algorithm ................................... 32
CONTENTS

3.7 Post processing ................................................................. 40
  3.7.1 Structurize ............................................................... 40
  3.7.2 Application ............................................................... 41

4 Implementation .............................................................. 43
  4.1 Framework ................................................................. 43
    4.1.1 Environment .......................................................... 44
    4.1.2 Algorithm ............................................................. 44
  4.2 UML diagrams ............................................................ 51
    4.2.1 Class Diagram ......................................................... 51
    4.2.2 Sequence diagram .................................................... 58
    4.2.3 State Diagram ........................................................ 61
    4.2.4 Use case diagram ..................................................... 64
    4.2.5 Deployment diagram .................................................. 67
  4.3 Usage ..................................................................... 68

5 Results ..................................................................... 73
  5.1 Testing and analysis ...................................................... 73
  5.2 Results ................................................................ 74

6 Conclusion ................................................................. 78
  6.1 Observations .............................................................. 78
  6.2 Future improvements and extensions ................................. 80
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSP</td>
<td>Generalized Sequential Pattern</td>
</tr>
<tr>
<td>FP Growth</td>
<td>Frequent Pattern Growth</td>
</tr>
<tr>
<td>HMI</td>
<td>Human Machine Interface</td>
</tr>
<tr>
<td>PC</td>
<td>Personal computer</td>
</tr>
<tr>
<td>PCA</td>
<td>Principal Component Analysis</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>Regex</td>
<td>Regular Expression</td>
</tr>
<tr>
<td>CSV</td>
<td>Comma Separated Values</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>DUC</td>
<td>Device Under Test</td>
</tr>
<tr>
<td>KDD</td>
<td>Knowledge discovery from data</td>
</tr>
<tr>
<td>EB</td>
<td>Elektrobit Automotive GmbH</td>
</tr>
</tbody>
</table>
Introduction

Elektrobit Automotive GmbH (EB) is an automotive software firm which has various departments to deal with automotive domains like architecture, human machine interface (HMI), Navigation, Car Infrastructure, Driving assistance systems, connected cars and system testing. This research paper comes under the HMI department and its functionalities. The HMI department deals with all phases of development cycle of user friendly human machine interfaces which are up to date keeping in line with the upcoming trends. It is of great importance to the customers and the manufacturers to make decisions and give worth to their investments. EB helps produce reusable HMI components and also have scalable solutions in order to cope up with the new requirements of next product generation. This provides a better user experience and thus integrates all innovative ideas on all devices.

1.1 Motivation

Elektrobit Automotive GmbH supports all phases of development cycles for HMI. This paper concentrates on the system testing phase of the HMI’s and the consequent error handling. Developers develop user interfaces for different functionalities like navigation, telephone calls, media player, radio etc. These interfaces are tested on different targets (devices used to test the functionality) and the results are captured. This thesis is related to analyzing the log messages of these results. HMI tests are being carried on a regular basis. The infotainment system is connected to the test PC and the test cases are executed on the host PC where the device driver is installed. Developer can give inputs by pressing button or navigating through the hardware kit which should output the desired behavior. Reports of these tests activities produce a huge background trace
log which contains details about all the functions executed. Now what is a trace log? When writing code, developer outputs a trace message after execution of a particular logic. The traces contain messages to check if the preceding code has been implemented successfully or not, whether a particular data is present or not, to check if a connection is established or not, etc. The program flow can be monitored by checking the trace log since it has trace messages coming through various channels, sources or threads of the source code. But these messages cannot be directly interpreted by merely reading it. It needs to be analyzed by the developer since some messages can originate in components that are not within the HMI applications and can therefore not be interpreted reliably by an HMI developer and some are written by the coder himself.

Figure 1.1: Trace log file view

Figure 1.1 shows the structural view of a trace log file. It contains millions of lines of trace messages in detail for a small test case. It contains different columns like ‘PacketID’ which is unique for a particular message. But there are some multiline messages which have same packet ids in consequent rows. ‘SessionID’ describes the session number of the test scenario whereas ‘LoggerTime’ and ‘TraceTime’ are timings at host PC and test PC respectively. Next column is ‘Channel’ which shows from which source code function the message has come from. ‘Source’ column defines the thread which was used for that implementation. ‘Level’ is one of the most important columns used for analysis. It shows 6 different status of the log messages like: trace, debug, info, error, warning, fatal. They are given defined priorities like fatal being the highest and info being the lowest, with the help of which further analysis can be done. But sometimes fatal or error can be normal whereas trace level can be dangerous depending on the scenario and hence this is a reason that error analysis cannot be completely automated. The next column is ‘Size’ which shows length of the messages in bytes and the most important column is ‘Message’ on which different data mining and pattern recognition techniques will be carried out to find the normal and the abnormal messages. The trace logs need to be analyzed and pre-processed so as to yield good error prediction.

In the automotive infotainment system, while analyzing bug reports, developers have to spend significant time on reading log messages and trying to locate anomalous behavior before identifying its root cause. The trace logs are too large to analyze it manually and too unstructured to analyze it automatically. [1] The trace log file contains millions of
lines of trace messages, many of which are just informational texts of less importance. The log messages need to be viewed in a Traceviewer tool to read in a human readable form and have to be extracted to text files by applying manual filters in order to further analyze the behavior. A new developer would take weeks to analyze the trace log since it is tough to identify the normal messages and there is no direct comparison available. The usual developers would also take a few days or hours depending on how big or small the error is. But it is definitely a time consuming activity which leads to meaningless effort of scrolling through the log files. There are few temporary user defined techniques (described in section 2.2) used for error analysis, but those are not generic for all the trace logs. They are modified according to the need of each test case and hence there is no efficient solution available. Thus the need for finding some pattern recognition algorithm aroused which could remove the unnecessary data and have fewer amounts of data which could be meaningful for the developers to analyze manually.

1.2 Problem Statement

There is a need to evaluate machine learning/data mining methods which could potentially assist in error analysis. One such method could be learning patterns for “normal” messages. “Normal” could even mean that they contain keywords like “exception”, “error”, “failed” but are harmless or not relevant to the bug that is currently analyzed. These patterns could then be applied as a filter, leaving only truly anomalous messages that are interesting for analysis. A successful application of the filter would reduce the noise, leaving only a few “anomalous” messages. This is not the only method, or the focus of the thesis. The task also includes researching and evaluating different techniques that could be applied to reach such a goal. The real world scenario and raw data is provided in the context of a widely deployed automotive infotainment system.

1.3 Structure of thesis

The thesis report is broadly divided into seven chapters namely: Introduction, State of the art, Concept, Implementation, Results and Conclusion. Each chapter will explain in detail the technical and logical aspects of the system. Chapter 2 talks about the literature research and the candidate algorithms found. Evaluation of candidate algorithms and the final algorithm which will be used is also discussed in the chapter. Chapter 3 discusses about the algorithm and overall flow of the implementation. It has described all the pre-processing and structuring techniques which will be used in the implementation. Chapter 4 shows the framework of the designed prototype and their specifications in detail. It also describes the usage of each technique at every step. Chapter 5 depicts the final results produced after complete execution of the prototype. It discusses about the results produced and how good or bad they are. The chapter 6 concludes the thesis by pointing important findings and showcasing future scope.
State of the art

Looking at the format of the trace log which is quite unstructured, it was clear that data mining would be the area which could help in structuring the data and find useful patterns out of that. It looks like finding a needle in a haystack. Next section 2.1 explains the need of data mining and the methods associated with it. This chapter also covers evaluation of candidate algorithms 2.4 by a feasibility analysis of each algorithm in detail.

2.1 Motivation

In this age of Information Technology, every day there has been a massive increase in the data storage and it is doubling every few months. There emerged a need to extract knowledge from this huge set of unstructured raw data. Data mining is the field where you can extract meaningful information out of huge databases or repositories possessing detail information of the system by using intelligent approaches. Data mining is also termed as knowledge discovery from data as data mining performs intelligent searches to discover knowledge from a raw data set. Knowledge discovery from data is abbreviated as KDD [2]. Knowledge discovery process is an iterative sequence of the below steps:

1. Data Cleaning (eliminating noise)
2. Data integration (combining data from different sources)
3. Data selection (mark important data)
4. Data Transformation (perform operations to structure data)
5. Data Mining (using intelligent algorithm)
6. Pattern evaluation (identify patterns)

The first 4 points in the sequence are pre-processing techniques whereas others are actual data mining techniques. Depending on the data, post processing on patterns can also be done.

Data mining can be performed on various forms of data as long as the data is useful. Relational database, transactional database, database data, ordered/sequence data, data streams, data warehouses, spatial data, multimedia data, graph and networked data, web data etc. [2] Data mining functionalities show what kind of patterns can be mined through this process. The functionalities include:

1. Characterization and discrimination
2. Mining of frequent patterns, associations, correlations
3. Classification and regression
4. Clustering analysis
5. Outlier analysis. [2]

Out of the above, mining of frequent patterns and associations is the functionality that will be focused in this paper. The algorithms used under this technique are Apriori based algorithms: FP (Frequent Pattern) Growth and GSP (Generalized Sequential Patterns).
2.2 Existing alternative solutions

In order to ease the process of analysis of trace logs, HMI developers use some ad hoc user defined temporary alternative solutions. These solutions are specific to a particular test case and hence it is not a generic solution. Moreover, it requires additional effort to create such solutions by understanding what the need of the hour is.

Trace logs are viewed through a special tool called Traceviewer, which is a tool developed by Elektrobit partner: e.solutions. With the help of this tool developer can control the capturing of log by clicking buttons like Start log capture and Stop log capture. When enough log data is collected, developer can view trace logs or system logs in a human readable form. These traces are further filtered manually and extracted to different files (.txt, .log, .xls, etc.) for analysis which becomes very time consuming. Secondly, developers use xml files for easing the identification of specific required patterns. XML files contain manually added required pattern which is compared to the entire trace log file and thus expected result is found out. But this is not feasible in the cases where abnormal messages need to be found out. Some notepad macros are also used to check for patterns manually. The solutions used are not reliable, generic, adaptive, scalable or reusable. Hence, the need to find some alternative approach arrived.

2.3 Literature research

Trace log files are reports produced out of software testing and contain valuable background information regarding test execution. Analysis of trace log is the need of the hour since it wastes a lot of time of developers and testers. Trace logs are unstructured huge dataset whose manual analysis proves time consuming and frustrating at times when there is a need to search any small problem or to check whether the code has executed properly or not. Many researchers from different domains are working to solve this problem. Let us have a quick look into what research has been done in various domain of text analysis.

Data mining has designed various algorithms according to the need of the analysis. Algorithms such as Apriori, FP Growth, Apriori Hybrid, Apriori All, and Generalized Sequential Pattern are included under data mining’s functionality of mining of frequent patterns, associations and correlations. This field of data mining tries to find meaningful patterns and association rules in the dataset and predicts the future log files for error analysis. Data mining Clustering algorithms include CLIQUE, MAFAI, CACTUS and PROCLUS which find similar group of objects and form a cluster to remove maximum noise out of the data file.

Natural language processing (NLP) is a famous domain which is used to interpret formal
language and literature and understand the meaning behind the statements. It uses basic methods of parsing, tokenization, syntactic analysis, semantic analysis, pragmatics etc. This field has vast concepts and implementations but it is not directly useful for error log analysis because error logs are system based messages which cannot be interpreted directly based on language literature libraries. The error logs need certain pre-processing, post which they are eligible to be analyzed through NLP. There exists a method of NLP called Sentiment analysis which analyses real time unstructured log data to interpret the opinion or emotion of the messages.

Machine learning is another field of analyzing data through intelligent approaches. Machine learning algorithms learn the data through a training set (supervised learning) or by itself (unsupervised learning) and then produces patterns which helps to find deviations from the normal execution of the process. These patterns are also useful for clustering or classifications tasks and ultimately assist in detecting errors if analyzed well. Principal component analysis is one such unsupervised machine learning algorithm which can be used as an anomaly detection technique for trace log files. Let us look into some of these findings related to the thesis and also understand their feasibility for analyzing our trace log files.

2.3.1 Sentiment analysis

Various algorithms are used to extract knowledge out of raw unstructured data. One such technique is Sentiment analysis [3] which is covered under Natural Language Processing. Sentiment analysis is process of using text analytics and knowing the opinion behind the written data by understanding the meaning behind the words. It judges the emotion of the statements thus helping to analyze the purpose behind those lines. Sentiment analysis is used for real time social networks such as twitter, eCommerce websites where it is important to know the opinion of public which can be judged by reviews and comments.

Unstructured raw data is collected, transformed, restructured and stored in databases or big data format. The baseline algorithm uses different techniques like tokenization, Feature extraction and classification using classifier like Naive Bayes [3] etc. Tokenization [3] step involves dividing and picking up important elements from the statements which affect the statement directly like hash tags in twitter, emoticons, date etc. Feature extraction [3] is actually judging the mood and attitude of the sentences based upon the affirmations or negations used. A Naive Bayes Classifier in this case, extracts the vocabulary and counts the existence of the affirmation or negation. It then counts the occurrence of those words used. Depending on further calculation of these counts the sentiment of the paragraph can be concluded. Different interesting factors such as mood, attitude, emotion and traits can be judged from sentiment analysis.[3]

Sentiment analysis is also used for analysis of software trace log data which is produced
by program debugging, development and testing. Software tracing being low level has huge amount of data which is hard to analyze and includes all types of messages, variable values, entry exit points, errors and unusual behavior. Natural language processing approach is used for such analysis where the data log is analyzed using data log entries. Data log entries contain a sentiment score which is the result of sentiment analysis. This sentiment analysis classifier uses sentiment lexicon model score [4], sentiment proximity model score [4] and source code parsing model score [4] to generate a sentiment score. Based on the scores, respective data logs are highlighted depending on their higher positive or negative scores. [4]

Inspite of this remarkable analysis technique, it cannot be used for analysis of our trace log file. The reason behind this is sentiment analysis analyzes the attitude or emotion of the data. But in our case, there are situations where error messages and unusual behavior of data can be treated as normal. This is because there may be some unnecessary modules in the function code which do not run successfully and it does not matter for the overall performance of the test. There are also situations where a thread is not executed, an update failed or a function giving output which is not as expected. Such situations do not always point to an abnormality but they can be a part of a normal run of the test case. There is a level marked for each trace log message. The level for messages include trace, info, debug, warning, error and fatal where usually trace, info is the lowest priority and probability of abnormality whereas error and fatal is the highest. But many a times, when there is a normal run of the test case, here normal means required procedure is completed and the functionality is as expected, there are a lot of ‘error’ level messages which does not harm the functional behavior of the test. Whereas in an abnormal run of test case, sometimes info or trace can be harmful and lead to error based on what the message is. It purely depends on the situation of the test case and functional code. Hence, our approach is not to find the meaning of the message but to have a normal run of test case and mark those data as normal. Later these messages can be compared to a specific trace log with a known injected defect. The algorithm should be able to detect the abnormality (i.e. injected defect) in this case.
2.3.2 Principal Component Analysis

Principal Component Analysis (PCA) based anomaly detection [5] is an unsupervised machine learning technique. It is generally suitable to detect anomalies in the console logs which comply with techniques used by PCA as discussed further. Let us see the working principal of this technique and in the later part of this section, its compatibility and applicability with our trace log analysis.

In general, PCA is used in a two stage detection method [6] where first stage includes frequent pattern mining and second stage is PCA based anomaly detection. It is used to find dominant patterns and discover the underlying anomaly. PCA requires pre-processing of the console raw log data in order to be utilized further. Pre-processing steps include log parsing and Feature creation. Log parsing scans the log data and detects the structure beneath the messages. This is achieved by dividing the messages into message type and message variable. Much noise is reduced when the messages are structured and unimportant details are removed. Log parsing groups the common log messages and creates a message template. This is done by applying a filter to the source code and extracting log messages. These messages are indexed and searched run time through the log messages to eliminate useless noise. Next step is Feature creation which creates 2 features: State ratio vector and message count vector. State ratio vector is calculated by identifying the state of the messages. Examples of state can be abort, failed, return value etc. A matrix is built based on the states. Dimension of this vector is the number of log messages that represent those states at a certain time period. Next feature is message count vector where a matrix is constructed based on grouping message identifiers or message types and creating a vector for each such group. Message identifiers are message variable which represents the object manipulated by the source code. Examples of message identifier can be packet id, file path, a file name etc. Message count vector is calculated for execution trace in single session. Dimension of this vector is the representation of number of messages in the coordinate system representing the respective message type or message identifier group. Any of these two features work well with PCA. [7]

PCA is used for high dimensional data sets which are scalable. PCA calculates small set of coordinates which are also termed as principal components. These coordinates are the representation of original data set on a smaller graph which identify variations of data over the time and thus help in understanding the deviations. PCA captures such patterns on a transformational matrix called $PP^T$, where P is built by the principal components derived from the training set of data. Abnormal component of the data vector $y$ is calculated as $y_a = (I - PP^T)y$, where $y_a$ projection of $y$ on the abnormal subspace. Based on these components, finally a squared prediction error SPE is calculated to detect the abnormal occurrences of messages. [7]
\[ SPE = |y_a|^2 \]

Vector \( y \) is abnormal if: \( SPE = |y_a|^2 > Q_a \), where \( Q_a \) is threshold for SPE.\[[8]\]

Thus, this two stage detection technique \cite{6} is where we find normal patterns by frequent pattern mining and less common ones with PCA based anomaly detection. PCA was experimented with the type of log data we need to analyze, but PCA cannot be used for our trace log analysis because of few fundamental reasons. The log parser method divides the message into message type and message variable to predict the structure. But in our case, no such type is fixed. The messages are unstructured since they do not follow any fixed pattern. Secondly, the messages cannot be grouped based on their states such as failed or aborted because these states do not conclude the message being normal or abnormal. In this particular trace log, the messages do not intend their literal meaning but it is a situational based identification of messages being normal or abnormal. Hence, such messages cannot be grouped and no vector can be formed. PCA can still be calculated by message count vector which is feasible. PCA identifies abnormalities from its matrix by plotting the high dimensional data onto a reduced dimension. This requires that the data should be ‘linear’ or ‘scalable’ which is not in this case. Our test case does not have a defined structure but it depends on the developers runtime input. The trace messages are always based on the test case which is user defined and fetches input runtime. Output messages are always variable giving dynamic, unstructured and non-adaptive data. Hence, a matrix cannot be formed.

PCA itself diverts from the analysis which our developers expect. PCA detects anomalies by calculation of squared prediction matrix. The output of a normal test run may have abnormality which is fine for the developers. What is important to find is what went wrong functionally which cannot be calculated by any mathematical model but by learning the normal runs. Hence, a supervised machine learning approach, which provides a training set for learning normal patterns, should be used in our analysis.
2.3.3 Data Clustering

There are various clustering algorithms designed for data mining. The main principal behind such algorithms are that they collect similar objects in the log data and group them to form different clusters. The clusters are calculated based on the points in the subspace. To determine their correlation, distances between those points are calculated mathematically using a distance function. Different clustering algorithms use different approaches to define the clusters. Points in the subspace may have different attributes which poses difficulty to calculate distance between them. Choosing a distance function thus is a complex task. Traditional clustering algorithms fail to support high dimensional data because dimensionality of data affects the calculation and representation of points to form clusters. When the dimension of data increases, it becomes impossible to calculate the distance between points in the subspace [9] [10]. Thus, clusters cannot be formed. [11]

To overcome this, there are density based approaches where the algorithm divides the subspace into regions. Dense regions are identified and thus form clusters from the regions. To support high dimensionality of data, various clustering algorithms were introduced such as CLIQUE, MAFAI, CACTUS and PROCLUS. CLIQUE [12] and MAFAI [13] are Apriori like algorithms [12] which generate candidates at each level increasing time overhead and runtime complexity of the whole process. CACTUS [14] being a fast algorithm, is susceptible to append long strings in the clusters thus not suitable for pattern recognition in log files. PROCLUS algorithm [9] uses a K-medoid method where K is number of clusters defined earlier. It cannot be accurately determined how many clusters a log file can produce, hence not suitable. [11]

Most importantly, clustering algorithms are suitable for generic log files. Basically, the algorithm counts the number of different words and clusters similar words calculating their frequency. Storing of such clusters is another overhead. It is assumed that there has to be a correlation between such frequent words. Such words are grouped and the support value is increased. The clusters then predict the normal patterns in the log files. [11] But the trace logs in this case are not based on the correlation of words. Rather they have to be analyzed in a behavioral manner. Normal messages are not based on the words the messages contain. It is more of a sequence based analysis which can identify the abnormality. Hence, clustering algorithms won’t be suitable for error analysis of our trace logs.
2.3.4 Apriori Algorithm

Ramakrishnan and Agarwal proposed Apriori algorithm [10] which mines frequent itemsets and produces association rules as output. Frequent item sets are the items in the database whose support value is above the user defined minimum support value. Support of an item is calculated by number of transactions that contains the item. Items below minimum support are discarded. Apriori algorithm works by finding candidate item sets in all iterations of finding frequent itemsets one at a time. It scans the whole data to find frequent itemsets each time and creates a candidate itemset considering only items above minimum support. It prunes the candidate itemset based on the Apriori property: the subset of frequent itemset must be frequent. Later a join operation is performed where the previous candidate itemset is joined with itself. When all possible candidate itemsets are created, association rules are inferred based on minimum confidence values. Association rules are simple if-then rules which show relationships between different items in the transaction. It shows how one item is associated with the other item. For example: If Y occurs, then Z occurs. Below is the generic Apriori algorithm:

1. Find the frequent itemsets ($L_k$): the sets of items that have minimum ‘support’
2. Generate the candidate itemsets ($C_k$) by joining frequent itemsets ($L_k$)
3. Generate association rules using ($L_k$) having minimum ‘confidence’, based on Interestingness like Frequency, Abnormality, Consistency, etc.
4. Rate the generated patterns manually. [10]

Apriori algorithm was the first to introduce association rule mining. It scans the whole database for candidate generation and thus creates runtime overhead and also increases the complexity and memory cost. Association rule being the most useful patterns, unfortunately are not very interesting in analysis of our trace logs. Association rule depicts a transaction, for example, if A occurs, then B occurs too. But in this case, it is already known that if a particular source or channel occurs, then corresponding type of message will surely occur. It is not required to analyze such transactions or what comes when. The interesting aspect is to understand whether a message sequence is normal or abnormal.

There are various versions of Apriori based algorithms being introduced to overcome the drawbacks of Apriori Algorithm. Some of them are FP Growth, Apriori Hybrid, Apriori Growth, etc. Out of these, FP Growth will be used to mine frequent itemsets in the trace log file because this algorithm scans the database only twice and no candidate itemsets are required to be generated. [10]

Another version of Apriori based sequential pattern mining is Generalized Sequential Pattern [15] which uses Apriori based or FP Growth based frequent itemset generation.
Using these itemsets, sequential patterns are mined based on a minimal support and confidence value. Sequential pattern is more useful than individual pattern mining because when sequences of messages are checked as compared to individual message check, sequences have a strong error analysis. If a file has silently dropped out or a process is killed in the background, it will be seen in the sequence mining whereas such an error can never be caught in candidate mining. Because if nothing has occurred in the traces, you cannot know it by comparing only individual messages against the frequent itemsets or association rules. But a sequence will always have fixed set of messages which can be compared to the incoming trace log files.

Generalized Sequential Pattern (GSP) [15] is a sequence mining iterative algorithm. It scans the data many times to generate frequent sequences. In the first pass of iteration, it finds the support for all the items in the data and thus learns the frequent items. The items below the minimum support are deleted. In the next pass, GSP keeps on generating frequent sequences of events and stops when there are no frequent sequences at all after scanning the whole data. The previous seed set helps to generate new frequent candidate sequences. GSP requires Customer and Time attribute to be able to associate sequences according to the user specified window size. The parameters window size, max gap and min gap works out very well to extract sequences from the data as per user requirement frame. GSP scales linearly according to the transactions present in the data. The execution time is faster as compared to Apriori algorithm. [15]
2.4 Evaluation of algorithms

Out of all the candidate algorithms found, the difficulties of pre-processing the data was observed in using many of the algorithms. This is because of the nature of the trace log file. Each algorithm is suited only for certain type of datasets. In the above sections, approach of all the candidate algorithms have been studied along with their compatibility details with the trace logs. Let us see a short tabular evaluation of all the techniques which are closely related to solving the problem of analyzing trace logs. Table 2.1 gives the feasibility analysis matrix of all the possible solutions:

Observing the feasibility analysis in table 2.1 which was designed after careful study and experiments of trying the feasibility to create input for them, Apriori based Generalized Sequential Pattern was the one most suitable followed by FP Growth algorithm. This is because the trace log files needs sequence of data in order to verify whether the functioning is normal or not. There are many traces of normal runs which are provided to study normal patterns of messages. Finding only frequent messages is not enough because behavior of the trace cannot be judged. Association rule is also not useful since it is not intended to know which message comes from which source or channel. FP growth would be useful for the analysis but only after executing GSP analysis. This is because FP growth algorithm produces only frequent itemsets. Once GSP clears out all frequent sequences, the trace logs still contain a lot of single messages which are frequent and which could not be a part of any sequence because such messages come individually not associated with any function flow. They come frequently from different source threads irrespective of the channels. Examples of such messages can be regular updates of the time zone, the timely hardware interrupts occurring irrespective of the functions executing, continuous updates of some processes, periodic calculation of the position of the car and of the routes, status messages of all functions in the infotainment system, etc. Natural language processing is not suitable since the trace logs cannot be interpreted merely by understanding the meaning of their messages. With such validations, a final approach has been designed for implementation of error analysis for trace log files.
## Evaluation of algorithms

<table>
<thead>
<tr>
<th>Domain</th>
<th>Algorithm</th>
<th>Usage</th>
<th>Feasible?</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural Language Processing</td>
<td>Sentiment Analysis</td>
<td>Analyzes the attitude or emotion of the data and predicts the opinion</td>
<td>No</td>
<td>Aim is not to find the meaning of the messages but to find a deviation of abnormal messages from the normal ones.</td>
</tr>
<tr>
<td>Machine Learning</td>
<td>Principal Component analysis</td>
<td>Detects anomalies by calculation of squared prediction matrix and finds deviation by plotting its graph</td>
<td>No</td>
<td>Need linear, scalable, adaptive and structured data set to be represented on reduced coordinate system.</td>
</tr>
<tr>
<td>Data mining</td>
<td>Data Clustering</td>
<td>Counts the number of different words, clusters similar words calculating their frequency. Clusters are used to find patterns</td>
<td>No</td>
<td>Our trace log messages are not based on meaning or correlation of words. Trace log needs to be analyzed by the message sequences.</td>
</tr>
<tr>
<td>Data mining</td>
<td>Apriori</td>
<td>Outputs frequent item sets and association rules</td>
<td>No</td>
<td>Association rules cannot predict the abnormality and the algorithm has high time complexity.</td>
</tr>
<tr>
<td>Data mining</td>
<td>Generalized Sequential Pattern</td>
<td>Outputs frequent sequence patterns within a time window</td>
<td>YES</td>
<td>Very useful to detect and remove normal sequences in order to find the abnormality.</td>
</tr>
<tr>
<td>Data mining</td>
<td>FP Growth</td>
<td>Outputs frequent item sets</td>
<td>YES</td>
<td>Has less time complexity with only 2 data scans as compared to Apriori.</td>
</tr>
</tbody>
</table>
To understand the implementation, first understand all the techniques used behind the prototype. This chapter deals with the conceptual and technical details of all techniques and also depicts the overall flow of the system through an approach algorithm. Based on the above descriptions, below is an overall view of the thesis highlighting where the focus lies.

Figure 3.1 summarizes the flow and focus of the thesis. Upper left corner shows the real HMI kit which needs to be tested. It is connected to a test PC in order to conduct different tests to validate its functionality and usability. The tests include switching on the device and checking whether the connection is established or not, pressing different buttons and checking their responses, navigating through various commands on the hardware screen, using radio, telephone or navigator and checking their functionalities. The Traceviewer tool is also connected through the test on the test PC. When the test is initiated, developer can capture the logs through this tool by pressing prescribed buttons like start log capture and stop log capture. Now there is a huge set of trace log which will be extracted to a text file. Here the main work starts. Different solutions need to be found out which in conjugation with manual analysis will assist in error analysis. It should be noted that manual analysis is a must here because all of the trace massages are not standardized yet and thus they do not follow a specific format. Hence it cannot be fully automated. A solution can be rated as good or bad depending on how much more data or noise it helps to reduce so that the manual effort is minimal.
3.1 Approach

The basic approach is to find frequent patterns from normal error free test runs and later remove these patterns from trace log under test. By this way, the remaining trace messages will need less manual effort which can be checked for abnormality. Focus is not to find the abnormality but to minimize the manual effort to find that abnormality.

Based on the feasibility analysis of the algorithms, algorithm has been thoughtfully designed to understand the flow of the implementation plan. There are two inputs for the whole processing. First one is the raw trace log file generated by error free test runs containing normal trace messages. Second is the standard library of messages which is extracted from the source code by applying a filter script. First and foremost check is presence of too many ‘HMI Dropped’ messages. If there are many such messages, the analysis is aborted. This message occurs when a function is not executed due to unavoidable system load on the target. There are basically two processes involved: Creating a training set (as shown in figure 3.2) and Error analysis (as shown in figure 3.3).

The first process, as shown in figure 3.2, starts with removal of noise and basic preprocessing of trace logs. The trace log is structured and free of noise. On the other hand,
3. CONCEPT

3.1. Approach

Figure 3.2: Concept algorithm: Create training set

The standard message library is converted into regular expressions in order to remove the dynamic part of the messages. These regular expressions are matched on the trace logs and this reduces huge amount of data. The trace log is indexed column by column so as to create unique ID’s for them. Since GSP algorithm is being used, the trace file needs
to be tweaked so as to create input for the algorithm. Input is created by transposing the messages by sorting the Source column and differentiating by TraceTime. Pattern recognition algorithm (here GSP) is executed in Rapid Miner tool which gives huge combination of pattern sequences as output. The output need to be post-processed in order to fetch unique meaningful patterns. After the post-processing a training set is created. It is compared to the previous training set to have same patterns. If the comparison does not result in a high match, this training set is stored after a manual grading. If the comparison is a high match, it is displayed to the developer.

The second process, as shown in figure 3.3, involves a new trace log file which needs error analysis. Since this file being a raw file, it needs to go through the same pre-processing steps that are performed on the training data which includes removing noise, structuring and sampling. Indexing and further steps are not performed since patterns are to be found on original messages in this trace log. The training set patterns are compared to the structured new trace log file for checking the amount of normal data present. After the matching process, we calculate the percentage of matched patterns and also the percentage of remaining manual effort. The smaller the manual effort is, the better is the result of the process.

Figure 3.3: Concept algorithm: Analysis of trace log
Trace logs contain raw and unstructured data. An approach has been designed to clean up the unnecessary data and then filter unwanted messages. The pre-processing techniques followed hereafter are noise reduction, removal of useless columns, filtering of useful channel sources and removing special featured messages. Later a special file ‘Message library’ is used to convert trace log messages in the form of regular expressions. This will ease the task of creating input for pattern recognition algorithm and also in assisting error analysis.

3.2 Pre-Processing Techniques

The most important pre-requisite for data mining is that the input data should be clean and well structured. It is as important to work on pre-processing techniques as it is to work on finding the right algorithm. Here, the raw data was provided after filtering it from the Traceviewer tool. It contained lot of useless data which need to be removed. Also the columns need to be separated to be able to use them constructively.

3.2.1 Noise reduction

Noise reduction involves removing unnecessary data which will not be useful for the analysis. In this case, the data file is fed with the $ delimitation in place of tabs to be able to separate the columns consistently. When done, it can be imported in to different file types like excel, csv etc. Out of all the columns imported, not all of them are useful for further analysis. Some of the unnecessary columns should be removed which include Session ID, Label, LoggerTime, Level and Size. An example showing the data contents of such columns is shown in table 3.1

<table>
<thead>
<tr>
<th>Session ID</th>
<th>Label</th>
<th>LoggerTime</th>
<th>Level</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
<td>21.08.2015 07:21:30.301</td>
<td>info</td>
<td>82</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>21.08.2015 07:21:30.304</td>
<td>info</td>
<td>132</td>
</tr>
<tr>
<td>0</td>
<td>–</td>
<td>21.08.2015 07:21:30.323</td>
<td>debug</td>
<td>33</td>
</tr>
</tbody>
</table>

The column session ID is not needed because all the tests are mostly executed in a single session. Label is an empty string and thus not useful. Out of two time columns, LoggerTime is removed because it is dynamic and hence further calculations become complex. Instead TraceTime always start with the same value since this is the device timer. It makes the calculations easier and the automation process generic. Level of a message is important because it shows what kind of message appears. Types of levels are info, trace,
debug, warning, error and fatal. These levels have priorities assigned to them where info is the lowest and fatal is the highest. But in this case of trace logs, normal test runs do have some error messages which are completely normal in that situation. Whereas sometimes, ‘info’ level messages in the error trace logs are not normal. Hence, one cannot judge a message from the type of its level. If that would have been the case, prediction of errors would be a simple filter. This is the reason why analysis of this type of trace log is tricky and is only associated with the message looks like. Last column eliminated is ‘Size’ as it does not really matter what length of message exists.

After this step, only columns remaining are Packet ID, TraceTime, Channel, Source and Message. Packet ID is the unique number for each message and is very important during giving input to any algorithm. TraceTime is important to calculate the time lags between messages and thus sort the messages based on it. It is also important to study TraceTime in the situation where a normal process would expect one minute to execute whereas it is taking more than 2 minutes, which clearly depicts the abnormality. Channel is the package of the function code from where the trace message originates. It is important to note the channel in order to understand which messages belong together logically. Source is the actual thread which executes the function and outputs the trace message. Source is one of the influencing factors to categorize messages in different groups. In fact, messages are sorted based on Source while creating input for the algorithm and using TraceTime to calculate the time differences between messages.

3.2.2 Filtering

Once the basic noise removal is performed, it is essential to eliminate such data which are unnecessary to analyze. One such elimination denotes the channel types. There are quite many types of channel all of which come from different automotive domains like navigation, infotainment, telephone, Bluetooth, radio, HMI etc. Out of these, only HMI reports are interesting to analyze because we are testing HMI applications in the department. It would be beneficial to analyze all other channels too but their source code and specification are not present as they are being developed in separate departments in the company. Some components are even developed at different companies which mean we cannot test the validation test cases since we do not have access to their source code. Each component is performing own their test and bug analysis, hence the concentration for HMI application sustains. Keeping this in mind, the data is filtered for channels containing only hmi.App.*. There are 596 different channels present in the trace log out of which we need the traces only from channels hmi.App.*. There 111 hmi.App type channels. Out of 209457 lines of data, 33066 remain to analyze. This removes 176391 lines which calculate upto 84.21% of data reduction.
3.2.3 Sampling

After successful filtration and noise free data, sampling of the data is needed. It is known that each message has a unique Packet ID. Contradictory to this assumption, there are some messages found which have same Packet ID’s. After careful examination, it was found that they were same messages with a line break and thus continuing with the previous packet ID. These are termed as ‘multiline messages’. Mostly these messages contain long chain of numbers or coordinates, hence not very constructive for further analysis.

In the figure 3.4, Packet ID: 1.1622 contains huge long chains of numbers going quite wider and deeper. These messages are rather a chaos and can misguide the process of pattern finding. The best option is to retain only the first line of the message because it is enough to identify the message and match it to the message library. The intention is not to lose the Packet data. To further reduce the data, multiline messages are removed. Preprocessing_remove_multiline.exe is used to automatically search the Packet ID’s which are repeated more than once. It selects only the first line of message and discards all other messages. This contributes to large reduction of data giving out a quality data for pattern recognition. Filtering and sampling both the processes are automated and executed through a batch file which runs within 3-4 seconds. Manually this process would have taken 30 minutes or more. Filtering the data manually is easy but finding multiline messages is very difficult considering such a huge set of data.
3.3 Message library

Out of the two important inputs needed for creating training set, second input is the message library. Message library is a standard template of messages which are extracted from the source code by applying a filter script. These are the messages written by the developer while writing the function code which will definitely appear in the trace log file depending on what functions are executed. The standard messages do not have a fixed format but they do have a fixed concept. An example of message library is shown in figure 3.5.

The message contains a static and a dynamic part. A generic example of a message will look like: Trace.log, (“Static part of the data is: %1, %2”), parameter_1, parameter_2. Here %1 and %2 are the dynamic part of the messages which are replaced by parameter_1 and parameter_2 at runtime, respectively. The message library clearly states the static and dynamic part. But it is difficult to identify these parts in the trace log file. Hence, to identify the standard messages present in the trace log file, it is necessary to generalize the message library. For this reason, the standard message library is converted into regular expressions. With the help of these wild card characters, messages will be replaced by the regular expressions and the dynamic part of the data will be removed from the trace log file keeping only unique messages in the trace log. This process further reduces the amount of data. In the trace log file, there are certain system messages or messages which are not present in the message library. Such messages will be handled by the indexing process explained in the next section.

3.3.1 Regular expressions

A regular expression describes the syntax structure of a log message and can then be used for Pattern Matching, even when the dynamic parts of the messages change. Converting a textual sentence in a regular expression is a process of generalizing the texts and creating generic patterns in order to identify the data. Regular expressions can be thought...
of specialized notations which are patterns that are being searched for in the text files. In this case, trace log file need to be analyzed along with another file called message library which is a standard template of messages which occur in the trace log file. Regex is very beneficial in this case to find patterns in the trace log file. Patterns can be searched easily if all the messages in the message library are converted into regular expressions and then match them in the trace log file. When a training set of such regex messages is created and new trace log file is available to analyze, common patterns can easily be found by searching with regex messages. It will ignore the ever-changing dynamic part and will match with maximum messages, thus removing quite a lot of patterns. This leaves the trace log file with fewer messages which can be manually analyzed.

Wildcard characters are used to form regular expressions. Each wildcard character has a meaning associated with it with the help of which one can create generic patterns for many complex texts. Some of the examples of wildcard characters are asterisk (*), dot (.), question mark (?) etc. These represent number or type of characters which can pass in the presence of that particular wildcard entry. * represents zero or more number of any character allowed. So if the regular expression is *.csv, then any filename with an extension of .csv is allowed to pass. This allows reduction of data to a greater level. Regular expressions are also termed as ‘regex’. Most important use of using regex is validation. For example: Suppose a form is to be filled and some fields require specific kind of data. Phone number field would allow only numbers and not alphabets and also require only 10 numbers. Email address should contain ‘@’ and ‘.’ as compulsory characters. Email validation through regular expression is as shown in figure 3.6

![Figure 3.6: Validation of Email address via regex](image)

The example shown in figure 3.6, the first part of text is checked against special characters and any alphanumerical character set. The symbol ‘@’ is mandatory which will be followed by another set of alphanumerical characters including dash and dot. The last part of the text should begin with a compulsory dot ‘.’ character followed by any upper or lower alpha character.
lower case alpha character containing 2-6 characters only. Such mandatory validations can be checked by writing regular expression at the backend of the code. Whenever a value is entered, it will be checked against such regular expressions and thus the form is validated. Basically regex are mostly used for searching, text processing and validation.

Regex is an important method used in the field of Natural Language Processing (NLP), an area of computer science, to search texts and are generic representations to combination of strings. [17] NLP is a branch of interpreting language details. It is used to study string processing and verify the content of a file. To understand this from a computational point of framework, the texts are processed through regex patterns. This operation helps verifying the formatting and context of the document. There are different building blocks for creating regex such as optionality, repeatability, choices, ranges, complementation, special symbols and wildcards.[18] Applications of regex in NLP are phonology, morphology, information extraction, speech recognition, text analysis, etc. [17] Another application in the field of biology gave rise to Sequence mining algorithm with constrained pattern matching [19] where there are sequences which need to be verified to a desired structure. This problem is solved by creating a regex of combination of letters and applying constraints on the available sequences. Those sequences are validated only if they match the regex values.

3.3.2 Data optimization

Regular expression in this case is used for few reasons, one of which is data optimization. The messages present in the trace log file contain different version of same pattern because the dynamic part of the message is different most of the times. This causes excess memory usage and should be handled carefully. To integrate these messages in a single frame, such messages are converted to regular expression. This is done with the help of message library where already all the standard message templates are present. A list of all such messages is made and they are listed in a single file. This file is grepped with the trace log file so that maximum messages are replaced by their respective regular expressions. A huge amount of data is reduced here since repeated messages are gone.

![Regex Message Library](image)

In this case, the message library contain unique set of messages which are in the form Trace.log, (“Static part of the data is: %1, %2”), parameter_1, parameter_2. Here, %1 and %2 are dynamic values which will be passed runtime. The parameter value can be
anything based on the function code and return values passed. All the messages in the
message library need to be converted in a regular expression pattern so that the dynamic
part can be replaced through wildcards. When these messages are searched on the actual
trace log file, all the messages matching same type of pattern will be replaced by this
regular expression. Considering a file with messages such as “Static part of the data is:
dynamic1 and dynamic2”, Static part of the data is: textlog1 and textlog2” and “Static
part of the data is: error1 and error2”. All these messages will be replaced by the single
pattern “Static part of the data is: %1, %2”. This comprises 3 text messages into one mes-
sage. Imagine if there are 50 message of the same type, the amount of data reduction will
be quite huge. The advantage of doing this pattern matching is that there will be less data
available to find actual patterns and analyze them through pattern mining algorithm. As
the amount of data decreases, it becomes easier to find patterns. Another advantage is
all the messages are now in fixed formats and thus gives a structured data file. This pro-
vides data optimization and memory optimization. Data optimization is performed to
create structured input intended to be fed to pattern recognition algorithms. Later these
patterns are searched on the error trace log file to identify matching patterns. This makes
it easier to find the messages with varying dynamic parameter values. The comparison
complexity is less and the data scans reduce even more. This contributes to less process-
ting time, less memory overhead and less complicated approach.

In order to test the grep function, experiments were conducted with regard to pattern
matching using the tool called ‘Power Grep’ [20] which is an Open source tool. It is a
powerful Windows grep tool which uses regular expression as a key concept to search
through files. Besides searching for regular expression, it supports other actions of sim-
ple searching text, search and replace text, merge files, split files, search and delete text,
search file name, rename files, etc. It has a multiline field to enter your regular expressions
and browse your file location where the actions are needed to be performed. It supports
various file types like excel sheets, MS word document, OpenOffice files, PDF files, etc.
It has a good graphical interface to understand the usage quickly. The tool has a result
window which highlights your matched patterns which makes it easy to analyze your
conclusions. [20] Power grep helped the process of searching the regular expressions in
the message library and replacing them in the trace log file. The input file was message
library and the output file location was of the trace log file. The tool performs grepping
regex in few seconds saving a lot of time. This tool can also be run through command line.

After experimenting with the Power grep tool, it was clear that the search and replace
technique works well. But since the tool works on a free trial period and needs a license
later, the final prototype implements grep function internally through library functions
in Visual Studio. This process will be described further in prototype implementation
chapter 4.
3.4 Indexing and Structuring

![Diagram](image.png)

Figure 3.8: Indexing

The data at this stage is clean and structured. But the text messages cannot be directly passed as input as it is not intended to interpret the literal meaning of the sentences. Also, there are many repeated messages or source names which unnecessarily eat up a lot of memory. If they are replaced by an index, there will be very less unique data which can be grouped easily. Hence, a unique ID is assigned to all the source, channel and message data items. Each column is separated in a different file listing the column data. A unique query is fired which will list only the unique source names or channel names or messages and the duplicate values will be removed. This is done for memory optimization and structuring the data.

Each column data is appended with an ID such as Channel_ID_*, Source_ID_*, Message_ID_* for channel, source and messages respectively. (* here represents a unique index number). The unique numbers of source names are 35, channel names are 111 and message names are 4496. All the column data is assigned an index starting from 0 progressing sequentially. Once all the 3 columns are indexed, each of the indexed column data is traced back and replaced on the original trace log data. This process takes little time with the batch file since each column scans the data and replaces the ID number. So the whole data is traversed thrice (to replace columns channel, source and message). The batch file is implemented as a multithreaded function. The faster the processor of a computer, faster this process is. For Dual core processor, this process takes approximately 15 minutes. For Quad core, it would take around 7-8 minutes only.

3.5 Feeding Input

After indexing the column data of channel, source and message, replace them all in the original trace log file. Now the columns in the file are Packed ID, TraceTime, channel, source and message. The trace log data contains trace messages of 9 successful runs. That means the test case has been executed 9 time consequently one after the other. The test case begins with searching the destination address and initiating the navigation process. Each run begins with the same destination address: "GuidanceListener#startGuidanceTo([Gibraltar, GBZ, Gibraltar, Zentrum, 36°08′43.5″ N, 5°21′11.7″ W, PICNAV, posValid] )". This message is indexed with a unique number. Each run is separated so as to process
them individually for manipulating it in required input format. Each run is divided in separate files by passing the index of the above mentioned line as a separator parameter. There are 9 separate files of normal run trace log data.

GSP is used as a pattern recognition algorithm. GSP analyses horizontal sequences and outputs horizontal patterns. But the data has vertical sequences of messages. To convert all the messages in a horizontal sequence, basic transpose operation is carried out which takes a chunk of vertical column data and places them in a horizontal row format. But to decide which messages should be grouped and transposed, the messages are sorted on the basis of Source column. Source column is sorted in ascending order and the sorting is extended to message column too. Source is used in this sorting because source represents the thread which executes the function and writes the trace message. Channel could have also been used but source makes more meaningful since a single channel can contain different sources.

Once the sorting is done, the messages are grouped by evaluating TraceTime. The time difference between the messages is calculated and a condition is applied based on which the messages are grouped to a certain limited number and then the transpose takes place. If the time difference between messages is 10 milliseconds or less, the messages are grouped into a single chunk. If the time difference is more, the next message is not included into the group and the previous group performs transpose. The time difference can be varied in the source code. ‘10 milliseconds’ was decided as a condition because most of the operations require only 1-2 milliseconds. It is assumed that when the difference is more than 10 milliseconds, the message comes from a source (thread) of different function code. The intention is to group messages which execute same operation. That is when the sequence would be meaningful. Each sequence should contain messages which are logically coherent and belong to similar functional flow making the sequence relevant.

Another constraint while transposing the messages is the number of columns to be used by the transpose should be 15. This is a number which is concluded after many trial and errors. Initially the number was kept as many as possible according to the trace time difference, but those many columns were not accepted at the import phase. Experiments were performed to check test runs for different column numbers to see if it reduces more data. More the data columns, more problems it would give for processing GSP algorithm. The number was reduced to 30 which was accepted at the import level but gave problem when the algorithm was executed. GSP does not support for very long sequences of data. This is because it calculates the frequent items sets and then process sequences out of them. It calculates candidate sequence of length 2, length 3 and so on. It utilizes huge memory which gives ‘out of memory’ error. Secondly, the processor takes hours of time to process which fails ultimately. This can be understood in the next section where GSP
3. Concept

3.5. Feeding input

The process is explained in detail. It was after such experiments of trying longer as well as shorter sequences, that the number of columns was limited to 15. This is an optimized
value considering a balance between length of sequences and runtime. With this number of columns, GSP algorithm runs well maintaining less compromise on the length of the sequences. An overview of this process is picturized in the form of an algorithm in the figure 3.9.

3.6 Pattern recognition

The data is pre-processed and made ready for processing through the candidate algorithms which will capture sequence pattern outputs. There are various data mining tools such as R, Weka and Rapid Miner out of which Rapid Miner will be used in the further pattern recognition. Next sections take a deep insight into the data mining tool and the algorithms.

3.6.1 Rapid Miner tool

Among various data mining tools, Rapid Miner is most user friendly and leading data mining open source tool. According to a report presented by Chemnitz University of Technology at the International Data Mining Cup (2007) [21] [22], this tool ranks the best among the open source tools based on its applicability and technology. Rapid Miner can use all existing algorithms of WEKA data mining tool and it can open R data mining tools by a plug-in. Rapid Miner’s main advantages include portability, user friendly, graphical viewing of processes and results, codeless construction of processes, etc. Rapid Miner works well on almost all OS platforms like Windows, Macintosh, Linux and UNIX. It needs an updated Java runtime for its execution. In this thesis, Rapid Miner version 5.3.000 is used for our prototype implementation. There is an updated version 6.0 available which has a drawback of not being able to be executed through command line. This feature will be feasible for Rapid Miner 6.0 when a newer version is launched. Rapid Miner 5.3.000 can be configured to execute processes from command line and works well. [22] A basic view of a new process in Rapid Miner 5.3 looks as shown in figure 3.10.

Rapid Miner has different sections. It showcases the predefined and implemented operators or algorithms in the operator view section. The repository view consists of two important folders: data and processes which stores imported files or datasets and created process in rapid miner, respectively. Process view is where user can construct their new process by dragging and dropping different operators and joining them sequentially. These processes can be configured by changing their parameter values in the process parameter view seen on the right. Process description shows the description of the focused process to let the user know its usage as well as input and output format.[22]
Rapid Miner processes are viewed and constructed graphically. A backend code of process meta data is generated after constructing processes which is stored in the local repository. When constructing new processes, Rapid Miner has the ability to predict the compatibility of operators and their results. This feature allows giving compile time errors while building the processes. User can also test intermediate results of each operator by a breakpoint facility in the building blocks which is rare in other related tools. More than 500 operators are present in this tool which is grouped according to their functionality such as process control, utility, repository access, import, export, data transformation, modeling and evaluation. Rapid Miner supports import of various file types such as Excel, CSV, XML, binary file and various data sources such as Access, Oracle and Microsoft SQL server. [22]
3.6.2 Algorithm

Based on the evaluation of the literature research, there are two candidate algorithms found to be useful to search normal patterns in the error trace log. Each of the algorithms is described for their usability, requirements, configuration and execution methodology through Rapid Miner tool in the following sub sections.

3.6.2.1 GSP algorithm

GSP stands for Generalized Sequential Pattern algorithm. GSP works very well in regards to speed and memory consumption in Rapid Miner when given right input and parameter values. We have already seen why GSP is most suitable for finding patterns for our trace log file. GSP scales up linearly with the increasing data sequences. To implement GSP, the input needs series of data sequences in horizontal order. The messages present in our trace logs are arranged vertically. These need to be transposed horizontally in order to be used by GSP. To make this happen, the messages are sorted by Source column and then chunk of messages are transposed from columns to rows. They are grouped on the basis of time difference between the current and next message. GSP needs transactions, and we use this heuristic to generate transactions from a stream of log messages. If the time difference is more than 10 milliseconds, then the next message goes into next message group and the current message group is immediately transposed. The input file also needs to have time column and transaction or customer column. Once the transposed file is ready, a time column is inserted with manual sequential values starting from 1. Similarly, we already have a source column which is imported later as a customer column and the values it contains are considered as customer ids. Source is considered as customer because it is the medium from where the messages are originating. Once the file is imported in rapid miner, the message columns are marked as nominal attribute because the process later converts nominal to binomial for further processing in GSP. [15]

There are basically two processes required to be created to find the sequences:

1. Import CSV
2. Execute GSP.

Graphical block diagrams to view the parameter configurations for each processes is viewed further.

1. Process: Import CSV

The process, as shown in figure 3.11, Import CSV contains two operators: Read CSV and Store. Read CSV imports the file containing input data and Store operator saves the file in the local repository. The file paths are to be defined for both the operators in the process parameter section which is shown in the figure 3.12.
3. Concept

3.6. Pattern recognition

As seen in the figure 3.12, Read CSV parameter allows specifying the file path which needs to be imported in the local repository of rapid miner. Based on how the file is formatted, the parameter column separators can be configured to comma or semi colon or any other delimiter.

To assign user defined names and attribute roles, data set meta data information parameter is used as seen in figure 3.13. In our case, first column is imported as Time which is a requirement for GSP. The source column is later renamed to customer id in further pre-processing. The other message columns are imported as nominal attribute since GSP needs binomial values to find frequent sequences. These nominal values are converted to binomial by a nominal to binomial operator.

Store operator accepts an input parameter where you need to specify the path in the local repository of Rapid Miner where the input file needs to be stored. The path is to be entered in the field shown in the figure 3.14.

2. Process: Execute GSP

This process has two views since it contains a sub process. Overall view of GSP process is shown in figure 3.15.

In the figure 3.15, the sub process block consists of pre-processing operators to feed desired input to the GSP operator. The GSP operator outputs its sequences which is written in a text file and exported in the windows file explorer through the operator ‘Write as Text’. This operator is fed with the output path of the file in the parameter section which looks as shown in figure 3.16.

Sub process block involves several pre-processing operators as shown in the figure 3.17.
Retrieve test operator accepts the input file from local repository and is fed to the operator nominal to binomial after structuring the data through intermediate operators. Once binomial values are created, those are again pre-processed to be analyzed by GSP successfully. The time and source column are renamed as time and customer id in the Rename operator. In this way, both the processes Import CSV and GSP can be constructed.

The processes of importing the input CSV file and executing the GSP algorithm are created and saved in the local repository. These have been tested manually on the tool for
3. CONCEPT

3.6. Pattern recognition

Different results. GSP requires parameter values for Time window and max and minimum gap. These parameters help to find repeated sequences in a fixed window of messages based on Time column. To automate the process of GSP, we are importing CSV file through command line which is executed without the need of opening the tool. This can be achieved by passing following command lines in a batch file as seen in figure 3.18. Similarly, GSP process is also executed in a similar way through a batch file with command lines shown in figure 3.19.

An additional environment variable MAX_JAVA_MEMORY is set for a maximum memory say 1024 in this case. This is a java runtime memory required for calculation of finding frequent sequences out of the huge binomial values. Usually, this value (if not initialized) is set to 64 Mb by default in rapidminer.exe but this memory is not sufficient to process
huge data like the one in this case. It gives an ‘Out of memory’ error. Hence, it is necessary to initialize this value.

The execution of GSP results in frequent sequences with permutations and combinations of all possible sequences. But sequences are needed only in correct order. This means even a single gap in a sequence is not desirable. Hence, some post processing is required in order to find correct structure of messages. It is also required to find unique sequences i.e. subset of sequence should be removed to achieve less processing time for comparison of sequences in error logs. The biggest sequence is retained. Although during comparison, even if a part of sequence is matched, it is removed since that could be a subset of the sequence.

Figure 3.20 is an example of how GSP output looks like. Initially it gives sequences for 2 items (columns) with all possible combination of messages of that order with their corresponding support values which can be seen in the extreme left. Maximum item
The message id number depends on the biggest sequence in the input. These message ids are indexed values of the messages which can be decoded in post processing.
3.6. Pattern recognition

3.6.2.2 FP Growth algorithm

FP Growth algorithm is Apriori based algorithm which finds the candidate items in the data set based on a minimum support value. Apriori scans the data as long as all the frequent items are not found. This causes processing overload and increases time complexity. To solve this problem, FP growth was introduced. This algorithm scans the algorithm only twice to collect all possible candidate items above a minimum support. It output a series of frequent item sets which can later be used to produce association rules. [10] [23]

FP growth is required because GSP rules out all normal sequence patterns out of the error trace log. But it was observed that still some frequent messages were remaining. This is because some messages are not always a part of a sequence but occur individually in the trace log quite frequently. These messages need to be captured and removed as well. FP Growth algorithm stood right for this cause. Only frequent messages are required from FP growth. Association rules will not be required since those are already proved invaluable. After executing GSP algorithm, same set of input is needed for FP growth algorithm. The block diagram for FP growth algorithm looks as shown in the figure 3.21.

FP growth algorithm outputs frequent messages with increasing items (columns) which look like GSP sequences. But these messages are not actual sequences but are ordered in this way based on their support values. By keeping item parameter as 1, all single frequent messages can be gathered. FP growth is executed through batch file similar to GSP which can be seen in figure 3.22.
3.6. Pattern recognition

Output of single items in FP growth looks as in the picture 3.23. This output is post-processed to gain unique frequent messages. These messages need to be decoded to convert the index values in their actual messages to be able to compare them with the new error trace log.
3.7 Post processing

The methodology with which GSP and FP growth algorithms work and what output they produce has been studied. The main task of the thesis is not find the patterns but to use them in such a way which can help reduce the analysis effort of the error trace log files. The outputs produced by the algorithms are still in a crude form and need to be converted into a structure which can be useful to compare the patterns in the future. Also the patterns present in the output contain a lot of noise. Noise here does not mean unnecessary data, but data which is created out of combination of unique data. Individual and unique sequences are needed to be extracted out of all the combination of sequences. For this reason, post-processing has been conceptualized.

3.7.1 Structurize

First post process is to structure the output file. It contains some tool specific and timing information in the headers which can be removed. For GSP, two important functions are checked: 1. Order check 2. Subset check
GSP takes a message column and combines all sequences to it to match the support. [15] This results in thousands of combination of a single sequence. For example, if an input sequence is Message1, Message2, Message3 and Message4. If this is a frequent sequence, GSP output will try to fit this with all combinations which will look as follows:

Message1, Message2
Message2, Message3
Message3, Message4
Message1, Message3
Message1, Message4
Message2, Message4
Message1, Message2, Message3
Message1, Message3, Message4
Message2, Message3, Message4
Message1, Message2, Message3, Message4

Out of all the combinations, the last sequence is useful because is the biggest sequence and it is completely in order. Other sequences contain a gap between their messages. This is achieved through checking the order of all messages by checking their column ids and also checks whether the sequence is already a subset of the bigger sequence.

Similarly for FP growth, out of all the single messages with different column ids, unique messages are required to be identified. The idea is to rename the column ids to a similar name because at this point it is not required to know what column the message belongs to. Here, the single message is interesting and not its order. Once all the messages have same column id, it is easy to find unique messages by firing a duplicate query method on the output.

### 3.7.2 Application

To use these messages, first of all they need to be decoded from indexed value to their original message strings. This can be done by matching their indexes to the file where they were first indexed. Hence, it is important to save the message index text file. But only a single run output will not give plenty of sequence messages. The normal trace log file contains 9 runs of normal trace messages. This means a similar test case was executed 9 times and the trace message were captured in a single file. These runs can be separated by the start of the test case line. Already separate runs of data has been possessed through pre-processing. The idea is to combine and try out different combination of runs by concatenating the files. In the experiments that were carried out, different combinations were performed. This involved executing the algorithms on single runs, combining 2 runs each, 3 runs each, 4 runs each and executing all of the runs at once too. This gave quite a variety of patterns and all of them were combined into a single file which is called
a training set. The training set also need to go through a small processing to eliminate
duplicate sequences. So, 2 training sets of each algorithm are ready to be tested on the
error trace log. These training sets will be constant unless there are no major changes
in the code and consequent trace messages. Whenever a new trace log is produced this
training set can be directly compared to it, giving out much less amount of data from the
trace log, most of which is abnormal. The result of pattern mining on new error trace log
can be seen in the Results chapter 5.
Based on the above pre-processing techniques and data mining algorithms, a prototype has been designed. This prototype is implemented as a console application which consists of various batch files accepting parameter inputs for automatic execution. The implementation fulfills the need of pattern finding in the unstructured trace log file. This prototype can be further developed to an automated trace log analysis software tool which can be integrated in the testing environment. Further sections provide a technical insight in the implementation details.

4.1 Framework

To experiment with all the processes explained in the concept chapter, a small prototype has been implemented. This prototype contains several batch files. Similar processes have been grouped into single batch file. Each process is an .exe file which is called in a batch file and desired parameters are passed. Batch file was chosen over designing GUI, considering its flexibility and speed in development while still remaining user friendly and portable. Another advantage is the developer can change the parameters according to his needs and execute the processes with changes in the sequence of execution as well. It is a flexible framework with the name of the batch files describing their intended operations. Complete flow of the framework has been described through modular algorithms in order to understand the processes individually.
4. IMPLEMENTATION

4.1 Framework

4.1.1 Environment

The prototype has been designed on Windows 7 platform with Intel I5 2.2 GHz dual core processor, 64 bit operating system and 4 GB RAM. For running this prototype, at least 500 MB of memory should be free for computation. Visual Studio 2010 was the integrated development environment (IDE) used. The prototype is implemented in VB.NET programming language. For execution of the prototype, .NET framework 3.5 must be installed in the computer. Microsoft Access 2010 was used to convert Message library into regular expressions. Power Grep tool was used to run experiments with the replacement of regular expressions, indexing and pattern matching. These functions are now implemented by using regular expression libraries of Visual Studio 2010. Rapid Miner tool was used to implement pattern mining algorithms such as GSP and FP Growth. The tool needs updated Java runtime (Version 7 build 1.7.0 at least) for its execution. The more the memory and cores available in the computer, the faster is the batch file processing of the prototype.

4.1.2 Algorithm

Basically the framework is divided into two main sections: Creating Training sets and Error checking which are further divided into individual sub-sections. To outline the framework with an algorithm, a modular approach is followed. This means each sub section will contain the sequence of batch files and their internal .exe file being called in the respective batch files. Wherever necessary, the algorithm will highlight some important inputs provided to a process and their outputs produced, both represented by a dotted line.

I Creating Training set

Figure 4.1: Creating Training sets

The first section of the algorithm, as shown in figure 4.1, contains 4 sub sections which are followed in a sequence: Pre-processing, create input, execute algorithm and create training set. The whole process is started with the input Normal Trace log file.
Internally, another input called Message library is needed which will be shown in the individual algorithm of the sub section. Take a look into each sub section’s algorithm.

i Pre-processing

![Diagram of Pre-processing]

As shown in the figure 4.2, two main inputs are required for allowing execution of pre-processing techniques. Normal trace log file is the one which contains trace messages which are produced out of several successful runs. The messages in this file are used to find normal patterns of messages which will be useful in the error checking of new trace log file. Hence it is termed as normal trace log file. First and foremost it undergoes basic noise removal by calling remove_noise.bat file which contains calling several .exe files. The next process needs message library as its input and is executed by calling remove_dynamic.bat file which contains methods to remove the dynamic part of the regex messages. The next process called by indexing.bat file indexes all the columns individually giving out output files channel_indexed.txt, source_indexed.txt and message_indexed.txt. These files are again used to index the columns in the normal trace log file by calling index_log.bat. This process gives out a pre-processed, noise reduced and
ii Create input

The indexed log file produced by previous methods is given as an input in this process 4.3. The next step separates each run from the trace log file and outputs 9 separate run files. This is done by calling separateRuns.bat file. The next process called by transpose.bat creates actual input data by sorting all messages by source column and creating message groups based on time difference. This is repeated for all the run files. Concatenate.bat file accepts user defined input and concatenates the run files to create combined input. This depends on what input the user wants to test, hence the flexibility is given. Generate_indexed_run.bat appends an index column to the whole input file. This column when imported in rapid miner tool is used as Time column which is a prerequisite of GSP algorithm.
iii Execute algorithm

Once an input file is created, next sub section concentrates on execution of the data mining algorithms. As shown in figure 4.4, 2 algorithms are executed: namely GSP and FP Growth. The commands used to execute these algorithms are directly configured in their respective batch files. RapidMiner_ImportFile.bat imports the input file in the rapid miner tool. The processes are already created in the Rapid Miner tool and saved in the local repository. Import process is executed only once because same file will be used for both the algorithms. RapidMiner_ExecuteGSP.bat executes GSP through command line arguments in the batch file. It outputs a file which is processed by calling postProcess_GSP.bat. This process finds the sequence and subsets of the patterns and thus creates unique sequential patterns. RapidMiner_ExecuteFPGrowth.bat executes FP Growth algorithm to output frequent itemsets. Patterns from both the algorithms have been collected at this stage.
iv Create training set

As seen in the figure 4.5, the patterns created out of execution of the algorithms are combined together to create a training set for each algorithm separately. Once the patterns’ files are concatenated duplicate values are removed to achieve unique set of patterns. This creates a final training set where the entire data is indexed. It can be converted to text in the same process but to lessen the processing time, it is done in the error check processes.
II Error checking

The second section of the algorithm figure 4.6 includes two main sub sections called pre-processing and error check. Two main inputs are required in this section. The inputs are a new error trace log file and an indexed training set created in the last process. Pre-processing here is similar as in the section II but with less operations.

![Figure 4.6: Error checking](image)

i Pre-processing

The method of pre-processing, shown in figure 4.7, is a subset of training data’s pre-processing method presented in previous section’s algorithm. But it is still shown here to highlight different input required and output produced. Instead of normal trace log, here new error trace log file is pre-processed only to remove the noise. No other pre-processing like indexing is carried out here because it is
4. IMPLEMENTATION

4.1. Framework

required to find normal patterns in this file by comparing its messages with the normal patterns of messages collected in our training set.

ii Error check

![Error check diagram](image)

As seen in figure 4.8, the noise free error trace log file is now ready to be analyzed for normal patterns through our automated processing methods. ErrorCheck.bat automatically calls the indexed training set and searches for those messages in the error trace log file by referring to the message_indexed.txt file, message library and the indexed training set. The normal sequence of messages when found is removed from the error trace file one by one. This error check is executed for GSP and FP Growth training sets. The output of these checks is a file which does not contain normal patterns. It may still have some normal messages which were not a part of the training set along with some abnormal messages. This file is further given for manual analysis which leads to finding of the error or abnormality. The main task of the thesis is to separate maximum amount of normal messages from the error trace log file so that it is easier for the manual analysis to find the abnormality.
4.2 UML diagrams

This section has been designed to view the built prototype from different technical perspectives like static view, behavioral view, data flow and its usage. Such views with technical details can best be shown by UML standard diagrams. UML (Unified Modeling Language) diagrams model the software architecture of a system and are used mostly during requirement gathering stage to note the specifications of the system. It also helps to identify risks at an early stage which prevents errors at a later stage. There are various types of UML diagrams which are basically divided into two kinds: Static representation which depicts the static view of the system (for example Class Diagram) and behavioral representation which shows the data flow or process flow carried out in the system (for example Sequence diagram). For the prototype in this case, there are five suitable UML diagrams which can best represent our system and its specifications. Those are Class diagram, Sequence diagram, State diagram, Use case diagram and Deployment diagram. [24] The prototype implementation is represented with the help of each of these UML diagrams (wherever suitable) and can be seen in the further sections.

4.2.1 Class Diagram

Class Diagrams are static representation of the software system. It is a replica of the underlying logic code behind the processes. Class Diagram is necessary for displaying the objects and elements involved in the prototype. It also describes the relationships like dependency, associations, generalization, etc. that the objects share. Each class shown in the diagram represents a single element which executes the process. Each class has a class name, attribute names and methods or operations associated with it. To look from a modular approach, I have divided the class diagrams into sequential processes. Each process will describe its own class diagram. The processes involved in the prototype are:

I Pre-processing
II Usage of Message template
III Tokenization
IV Generate input
V Import file
VI Execute GSP algorithm
VII Analysis

Out of the above mentioned processes, process named Import file and Execute GSP algorithm are executed through Rapid Miner. The block diagram associated with these
4. IMPLEMENTATION

4.2. UML diagrams

Processes have already been seen. Take a look into class diagrams of each process individually.

I Class Diagram: Pre-processing

Classes in the class diagrams are built logically based on their functions and they are not actual names of any classes. The main intention is to show the static view of the source code or the prototype. Keeping this in mind, class diagram seen in the figure 4.9 possess 4 classes. Most of the classes have generic attributes: delimiter, input_path, output_path, input_array and output_array. All of the classes in all the processes involve generic processes read_file() and write_file() since it modifies the trace log file each time. First class ‘Create Delimited File’ accepts the raw input data and injects $ delimiter in the file in replacement to tabs. There are inflexible tabs in between columns because of which the columns cannot be properly imported. Hence a delimiter is forcefully injected quite a number of times until all tabs between the columns are gone. It also takes care of the $ symbol present in the messages and avoids the data corruption. It uses methods replace_tabs() and split_columns() in order to execute the process. This file is given as an input to the next class ‘Remove columns’. Hence, the import relationship is shown between both the classes. As the name suggests, remove columns eliminates the unnecessary columns which are not useful in further analysis using methods remove_column() and split_column().

Next class ‘Filter Channels’ accepts a channel filter and executes it using regular expression. It sustains channels associated with HMI department only. The class uses methods select_column() and apply_filter() for execution. The last class in this process is ‘Remove Multiline messages’ which accepts PacketID as input parameter and searches common values. It uses method comparePacketID() and remove_line() to search for multiline messages and remove them sustaining the first line. The classes ‘Filter Channels’ and ‘Remove Multiline messages’ are dependent on the ‘Create Delimited File’ class.

II Class diagram: Usage of Message template

The class diagram in figure 4.10: Usage of Message template represents 2 classes: ‘Generate regex’ and ‘Remove Static data’. Both the classes contain all the basic attributes and methods. Additionally, remove static data class has 2 more attributes: replaceMode and executeMode. The attribute replaceMode is important because it decides the mode of replacement as regular expression or replacement as literal text. Similarly, executeMode decides the mode of execution to utilize the processor which can be multithreaded execution or single thread execution, affecting the performance of the process. ‘Generate regex’ creates regular expression patterns for the standard message template or what is termed as message library. Hence it uses the functions generateRegex(), generateStaticData() where is deletes the dynamic part out of the
regular expressions keeping only static data visible. findDynamicData() which helps in finding the dynamic part of the text messages. Whereas remove static data class uses replaceData() and matchData() functions to find the matching text in order to replace it with its regular expression pattern. The relationship between both the classes is shown as dependent because without generating regular expressions it is not possible to replace the trace log file with the regex.
4. IMPLEMENTATION

4.2. UML diagrams

Figure 4.10: Class diagram: Usage of Message template

III Class diagram: Tokenization

Figure 4.11: Class diagram: Tokenization

The diagram 4.11 has 2 classes: ‘Index columns’ and ‘Index log data’. They both contain all the generic attributes and methods. Index column takes index as an attribute to assign indexes to the channels, sources and messages. The indexing is done by separating the columns source, channel and message by the function separateColumns(). To assign index it is important to remove duplicates entries if present. removeDuplicates() function removes the duplicate data. Last function assignIndex() as per the name of the function, assigns index to each column data. ‘Index log data’ class imports the files from ‘Index column class’ in order to take each column and their respective index and replace them in the trace log file. This class uses the methods findPattern() and replaceByIndex() to search for the desired data and replace it.
by mapping it with the correct index.

IV Class diagram: Generate input

The class diagram 4.12 has got 3 classes namely separate log runs, create input run and join runs. Each class has all the generic attributes and methods with specific individual methods. ‘Separate log runs’ class has an attribute named ‘separator’ which has the message ID that divides all the runs individually. findSeparateRow() uses the attribute ‘separator’ to find the starting point to divide the row. It divides the run data with the help of the function ‘splitRuns’. This class is imported by the class ‘create input run’ which accepts a columnNumber and rowNumber as input to create an input file. This is done with the help of the transpose function which uses 2 decision
making functions before executing the function. The first check function is calculateTimeDifference() which calculates time difference between consecutive messages to create a message group. The next check function is calculateColumnNumber() which checks how many messages are included in the message group. The number cannot exceed a threshold value to keep the data optimum which can be processed by the pattern mining algorithm. Next class 'Join runs' takes input parameter 'run_number' from the user according to how many runs it wants to combine to create a desired input file. A parameter is accepted dynamically for the user to do trial and error of the input data to obtain optimum result. Concatenate() function joins the specified number of runs.

V Class diagram: Analysis

![Class diagram](image)

The last class diagram 4.13 of the process analysis involves only 2 classes namely ‘Post Process’ and ‘Pattern Recognition’. Post process class accepts the patterns produced by the pattern mining algorithm and creates unique patterns. It uses checkPatternOrder() and checkSubsetPattern() functions to check whether the sequence patterns are in right order and evaluating presence of subset of the patterns. The patterns are structured by the function removeNoise() and finally splitPatterns() function is used to create unique patterns. ‘Pattern Recognition’ class has pattern and static_message as attributes which will be used to analyze the new trace log file for its abnormality. It takes the new trace log file and executes the function findPattern() and findSequence() both to ensure that the found patterns are in sequence. findStaticMessage() is used to find the message in the new trace log file which should be
replaced by the standard patterns. Finally replacePatterns() is used to eliminate the repeated pattern sequence data which is considered to be normal. Remaining data is given to the developers for further analysis. Relationship between both the classes is one to many and one to many. This is because the post processing can find any number of patterns in the new trace log file in the class ‘Pattern Recognition’ and any number of patterns founds can be post processed by the class ‘Post Process’.
4.2.2 Sequence diagram

Sequence diagrams are used to denote logical flow of the system. It represents the dynamic behavioral view in contrast to class diagrams which represents static view. By viewing a sequence diagram, process flow and data flow of the whole system can be learnt. Each element has its own lifeline and activation box for the time it is active in the process. An important aspect to be noted is sequence diagram depicts mainly the sequence of the process flow rather than describing what the actual process is.\cite{24} \cite{25} In this case, the system has been divided into 2 main processes which have their own individual sequence diagrams. The processes are ‘pre-processing’ and ‘pattern recognition’.

I Sequence diagram: Pre-processing

In the pre-processing diagram 4.14 the main elements are Trace log file, Message library, Noise removal process, Structurize data process and Tokenization. Pre-processing sequence diagram shows first process of creating a $ delimited file in order to get rid of unnecessary tabs and spaces. Hence, it is executed by the process ‘noise removal’. Next process again executed by the same process is removing of unwanted columns which would not be required for further processing. Trace log file is then pre-processed by ‘Structurize data’ element in the diagram to execute the processes of filtering channels and removing multiline messages. All the process return the noise free and structured file back to the trace log file element. Next process is initiated by ‘Message library’ element to convert all of its messages into regular expressions and get a regex patterned file. These regex need to be replaced in the trace log file. Hence trace log file initiates the replacement process. Structurize data return the final regex replaced trace log file. After finishing with the basic pre-processing, further step is to tokenize the data. Hence next processes are executed by ‘Tokenization’ element. The trace log file needs to index its columns separately. This process returns channel_indexed.txt, source_indexed.txt and message_indexed.txt files. Now the trace log file needs to replace all its important data with their indexes in order to create tokens. The final process returns the indexed trace log file which is ready to generate input.
Sequence diagram: Analysis

Elements in the analysis sequence diagram 4.15 include user or a developer, trace log file, creating input process, mining algorithm (GSP in this case) and analysis process. First process starts with the process of separating run data which is executed by 'Create input' element in the diagram. This process returns separate run files such as run1, run2, run3 and so on until run 9. This is because the trace log files contain log messages of 9 normal runs. It is necessary to split the run data so as to individual process the files. Next process of concatenating input files is initiated by the user or
4. IMPLEMENTATION

4.2. UML diagrams

The developer (in this case). The developer passes a parameter of how many runs should be concatenated. Accordingly the return file consists of user defined concatenated input run data. This data is passed to the next element ‘Mining algorithm’. This element imports the input data file whose path is specified by the developer. GSP i.e. the pattern recognition algorithm is executed which returns GSP output file which contains sequence patterns. These patterns need to be analyzed and so the last element ‘Analysis’ post processes the output patterns and returns unique patterns to the developer. The developer then analyses the patterns in the new trace log file to find frequent patterns to support in error analysis. This final process returns a pattern analyzed file which includes elimination of matched pattern sequences. The remaining data is fed to the manual analysis team which is not the part of our processes.

Figure 4.15: Sequence diagram: Analysis
4.2.3 State Diagram

State diagram of a system represents mainly the objects, states and events associated with the system. A state of a system is an output after certain internal or external event has occurred on the object. Sequence diagram shows the behavior of the system by addressing the processes. Whereas in state diagram, it is not about which processes are executing but it focuses on what happens when certain process is executed. State diagram is also used to show forward engineering and reverse engineering. State diagram are also termed as State chart diagram, State transition diagram or State machine diagram. Each block of the diagram represents a state of the system. In our case the blocks show the state of the trace log file when it undergoes different processing techniques. [24] [25] To view state diagram in a modular approach, the system is divided into 2 processes similar to sequence diagram. The processes are ‘Pre-processing’ and ‘Analysis’.

1 State diagram: Pre-processing

![State diagram: Pre-processing](image)

Figure 4.16: State diagram: Pre-processing

The pre-processing state diagram 4.16 shows different states of the trace log file when it undergoes various pre-processing logics. The raw trace log is the initial state of the
file. Here the diamond shaped box is a checkpoint to ensure whether are there many ‘hmi dropped’ messages in the whole trace log file. If there are a lot of such messages, it goes to the Abort state straightway which in turns goes to the final state. If there are not many of those messages, the system attains noise free trace log state once the event of removing unwanted data occurs. The events filtering and sampling of data makes the trace log structured. It is then converted into regular expressions by using an external message library input file which replaces all the trace logs by their regular expression patterns. The block representing the message library is a comment block which is used to give extra information. Then the process of tokenization achieves a state indexed trace log. Later these files are transposed and separated into n number of runs (9 runs in this case). Here the vertical narrow block is a fork which separates a single file into multiple file. In continuation, there is a join block which joins the runs into a single input file by the event of concatenation. The input file goes under various events of importing into the Rapid miner tool and executing GSP algorithm to give an output file of frequent patterns. This gains a next state of unique patterns after post processing events. The last state out of the whole set is a training set which is generated by comparing and combining all previous patterns. This is an end of the pre-processing phase because the training set is ready to be tested on various other trace log files to analyze the errors.

II State diagram: Analysis

The state diagram 4.17 for the phase analysis starts with the initial state of a new trace log file which is to be tested for having any defect or abnormality. This phase will conduct an analysis on the trace log file which will reduce maximum normal data out
of the file resulting in a minimum manual effort. The new trace log file undergoes same initial pre-processing which is for the normal trace log file. It attains the state noise free and structured so that it matches with the further pattern matching. Here, indexing process is not carried out because mining algorithm is not intended to be executed on this log, rather pattern recognition needs to be initiated by the already created training set of patterns. The set up messages are removed from the new trace log file because those are common in all test cases and need not be analyzed, rather analysis is much needed for the messages coming from the functional execution of the test case. The next state of ‘unmatched patterns’ occurs after pattern recognition event using the previously generated training set (as shown in the comment block). In this phase all the normal patterns have been removed. The leftover unmatched patterns are given for manual analysis which finds the error or abnormal messages if they exist and announce the file as OK or NOT OK. Here, the process of analysis ends.
4.2.4 Use case diagram

Use case diagrams are used to view the behavioral structure of the system. It has basic elements like actors, association, package, relationships etc. An actor can be a user or an external or internal application or a process which is involved with a certain activity. Each activity is shown serially and one or more actors are involved with the activity. Use case diagrams are mainly used to gather requirements of the system at an early period. Use case diagrams are used for forward engineering and after some modifications also for reverse engineering. Use case diagram is a good choice to represent a system which gives the architecture an overview of what things are needed to be implemented in the system and what risks or constraints should be taken into consideration. Use case diagram describe an activity involved in the system. Instead of viewing the whole system in a use case, it is always better to divide the system into their activities and design use case diagram for each one. [24] In this case, the activities are divided into 3 parts: pre-process, create training set and pattern recognition.

1 Use case diagram: Pre-process

![Use case diagram: Pre-process](image)

Figure 4.18: Use case diagram: Pre-process
4. **IMPLEMENTATION**

4.2. **UML diagrams**

Use case diagram, as shown in figure 4.18, for the activity of pre-processing involves 3 actors or processes in this case. Those are trace log file, message library and an index file. Each of the actors is involved in one or the other tasks. The actor trace log is involved in all the activities. The tasks included here are remove noise and filter and sample which involve the participation of trace log file only. The next activity of converting messages to regex is possible with 2 actors which are trace log and message library because the messages from message library are converted to regular expression which are then replaced with the messages on the trace log file. Next process involves the actor index file where in the trace logs are indexed and the runs need to be separated. Last activity of creating input involves only trace log file where functions like sorting and transpose are used.

II **Use case diagram: Create training set**

![Use case diagram: Create training set](image)

Use case of the activity create training set, as shown in the figure 4.19, involves only 2 actors namely pattern recognition algorithm and an automation tester. Pattern recognition algorithm is used in all the activities listed in the diagram. The activities listed are importing of the input file, executing a pattern recognition algorithm which is GSP in this case, find unique patterns out of the output of GSP patterns and finally building of a training set by comparing all the previous patterns and removing the duplicates. Automation tester is involved in all activities except executing the pattern recognition algorithm since this will be done through a data mining tool which is Rapid Miner 5.3 in this case.
III Use case diagram: Pattern Recognition

The use case diagram 4.20 showing activities in Pattern recognition process involves 3 actors namely a new trace log file which is to be analyzed for errors, message library and index file which were used in the first use case diagram of pre-processing and finally a manual tester to find the error. New trace log file is active with all the processes since the analysis is performed on this file itself. The activities start from pre-processing of new trace log followed by finding patterns in conjugation with message library and index files. The matched messages are removed and the unmatched messages are analyzed by the manual tester because the complete analysis cannot be automated as explained in the earlier sections. Lastly, after the analysis, errors are detected by the developer by manually viewing the reduced trace log file ending the pattern recognition and analysis phase.
4.2.5 Deployment diagram

Deployment diagram is a static representation of a system which includes hardware nodes along with software artifacts being deployed on that particular hardware node. [26] In the diagram 4.21, there are three hardware nodes involved which are HOST PC, TEST PC and the device under test (DUC). HOST PC consists of two more components carrying software artifact each. The first is the test plan component which contains all test cases. Another is the prototype implemented which is an outcome of this thesis work. It is called automated error analysis prototype. It is deployed on the same computer from where the test cases are executed. Tests are called from the HOST PC and passed on to TEST PC which connects and starts the device under test. The developer gives input to the device and the device returns its output to the TEST PC. These records are tracked in the trace log file which is captured by a Traceviewer tool on the HOST PC. Once a trace log is produced, the automated error analysis prototype is triggered and starts executing. It finally produces an analyzed trace log file. All normal message patterns have been removed from this analyzed file which makes it easier for manual analysis to find a particular error or a series of abnormal behavior.
4.3 Usage

The prototype is very simple to use as all the execution is based on calling the batch files. To configure the system, the folder containing all the batch files needs to be placed in any drive. The figure 4.22 gives an overall view of all the batch files in the framework used for execution. There are mainly 4 sections in this framework as separated by black horizontal lines and the intentions of those sections are noted besides them. There are batch files being called inside these batch files which are listed in detail in the table. The paths in the batch files should be updated based on the location of the respective input and output folder. Input folder should already contain the normal trace log file, message library, new error trace log files and few configuration files which are used in the processing. RapidMiner processes such as GSP and FP Growth should already be created and saved in the local repository. The path of this local repository should be updated in the batch files of executing GSP and FP Growth algorithms. Following table represents the purpose of the batch files in the framework and also the .exe files called inside the batch files. Table also
contains a column which shows what parameters are passed to the calling files. After studying these columns, it will be easy for a developer to use the prototype.
<table>
<thead>
<tr>
<th>Batch file</th>
<th>.Exe file</th>
<th>Parameters passed</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1_preprocess_delimitation_filtering.bat</td>
<td>Preprocessing_trace_log.exe</td>
<td>&lt;normal_trace_log_file_path&gt;&lt;new_output1_file_path&gt;</td>
<td>Removes unnecessary spaces and tabs and injects $ delimitation to separate the columns later.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Preprocessing_trace_log_3.exe</td>
<td>&lt;previous_output1_file_path&gt;&lt;new_hmidropped_output_file_path&gt;&lt;message column number&gt;&lt;Dropped Data: to sort only hmi dropped data&gt;</td>
<td>Separates HMI dropped data in a separate file. User has to check this to decide further processing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Preprocessing_trace_log_2.exe</td>
<td>&lt;previous_output1_file_path&gt;&lt;new_output2_file_path&gt;</td>
<td>Removes unnecessary columns: SessionID,Label,LoggerTime,Level,Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Preprocessing_trace_log_3.exe</td>
<td>&lt;previous_output2_file_path&gt;&lt;new_output3_file_path&gt;&lt;channel_name_to_be_filtered&gt;</td>
<td>Retains only hmi.app.* names channels</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Preprocessing_remove_multiline.exe</td>
<td>&lt;previous_output3_file_path&gt;&lt;new_output4_file_path&gt;&lt;literal_replacement_mode&gt;&lt;delimitation_value&gt;</td>
<td>Removes multilne messages</td>
</tr>
<tr>
<td>2_remove_dynamic.bat</td>
<td>grep_with_regex.exe</td>
<td>&lt;previous_output4_file_path&gt;&lt;message_library_file_path&gt;&lt;new_output5_file_path&gt;&lt;delimitation_value&gt;&lt;regex_replacement_mode&gt;&lt;multithreaded mode&gt;</td>
<td>Removes dynamic part from messages converted to regular expressions yielding only static part of the messages</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>grep_with_regex.exe</td>
<td>&lt;previous_output5_file_path&gt;&lt;configuration_file_path&gt;&lt;new_output6_file_path&gt;&lt;delimitation_value&gt;&lt;literal_replacement_mode&gt;&lt;singl e threaded mode&gt;</td>
<td>Performs some delimitation cleaning tasks</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3_preprocess_indexing.bat</td>
<td>Preprocessing_separate_columns.exe</td>
<td>&lt;previous_output6_file_path&gt;&lt;new_output7_file_path&gt;&lt;column_number&gt;&lt;delimitation_value&gt;&lt;literal_replacement_mode&gt;</td>
<td>Separates each column by using delimitation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Preprocessing_remove_duplicates.exe</td>
<td>&lt;previous_output7_file_path&gt;&lt;new_output8_file_path&gt;</td>
<td>Removes duplicate values to find unique values</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>indexing.exe</td>
<td>&lt;previous_output8_file_path&gt;&lt;new_output9_file_path&gt;&lt;index_prefix_name&gt;&lt;delimitation_value&gt;&lt;literal_replacement_mode&gt;</td>
<td>Assigns a unique index value</td>
</tr>
</tbody>
</table>

Figure 4.23: Parametric view of batch files(1)
4. IMPLEMENTATION

4.3. Usage

<table>
<thead>
<tr>
<th>Batch File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4_preprocess_index_log.bat</td>
<td>grep_with_regex.exe (Repeats for all columns: channel, source and message)</td>
</tr>
<tr>
<td>5_separate_runs.bat</td>
<td>separate_runs.exe</td>
</tr>
<tr>
<td>separate_run_files.exe</td>
<td>&lt;Previous_output10_file_path&gt;&lt;new_output11_file_path&gt;&lt;run_row_separators_file_path&gt; Creates separate run files</td>
</tr>
<tr>
<td>transpose.bat (Repeats for all runs)</td>
<td>Shift_delimited_col.exe</td>
</tr>
<tr>
<td>transpose.exe</td>
<td>&lt;Previous_output12_file_path&gt;&lt;new_output13_file_path&gt;&lt;delimitation_value&gt;&lt;number_of_columns_allowed&gt;&lt;source_column_index&gt;&lt;time_difference_allowed_minutes&gt;&lt;time_difference_allowed_seconds&gt;&lt;time_difference_allowed_milliseconds&gt; Transposes a message group from columns to rows</td>
</tr>
</tbody>
</table>

- 7_*_Create_GSP_Patterns_Set_Run_*_.bat calls concatenate_files.exe, generate_index_for_run.bat, RapidMiner_ImportFile.bat, RapidMiner_GSP_process.bat, post_process_GSP.bat and rename command.
- 7_*_Create_GSP_Patterns_Set_Run_*_.bat is used to combine different combination of runs for creating a combined GSP results training sets. Here run combinations used are: run12, run34, run56, run78, run1234, run5678 and run12345678.

<table>
<thead>
<tr>
<th>Batch File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenate.bat</td>
<td>concatenate_files.exe</td>
</tr>
<tr>
<td>generate_index_for_run.bat</td>
<td>indexing.exe</td>
</tr>
<tr>
<td>grep_with_regex</td>
<td>&lt;Previous_output15_file_path&gt;&lt;configuration_file_path&gt;&lt;new_output16_file_path&gt;&lt;delimitation_value&gt;&lt;literal_replacement_mode&gt;&lt;single threaded mode&gt; Replaces $ with comma to create CSV file</td>
</tr>
</tbody>
</table>

Figure 4.24: Parametric view of batch files(2)
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RapidMiner_ImportFile.bat</strong></td>
<td>Imports input file and saves it in the local repository</td>
</tr>
<tr>
<td><strong>RapidMiner_GSP_process.bat</strong></td>
<td>Executes GSP algorithm</td>
</tr>
<tr>
<td><strong>Post_process_GSP.bat</strong></td>
<td>Checks sequence order of messages and also checks for smaller subset present. Creates unique sequential patterns</td>
</tr>
<tr>
<td><strong>createTrainingSet_GSP.bat</strong> (repeat for GSP and FP Growth output patterns)</td>
<td>Concatenates all GSP/FP Growth output patterns files</td>
</tr>
<tr>
<td><strong>Grep_with_regex.exe</strong></td>
<td>Used only for FP Growth patterns. Helps structure the patterns to process further</td>
</tr>
<tr>
<td><strong>Preprocessing_remove_duplicates.exe</strong></td>
<td>Extracts unique sequential/frequent itemset patterns</td>
</tr>
<tr>
<td><strong>error_check_v1.exe</strong></td>
<td>Replaces index to original messages and creates a final training set</td>
</tr>
<tr>
<td><strong>errorCheck.bat</strong> (repeat for GSP and FP Growth training sets on new preprocessed trace log files)</td>
<td>Checks for frequent patterns on the error trace log and marks them as per the sequence number and pattern number to be removed later.</td>
</tr>
</tbody>
</table>

- **9_*_Create_FP_Growth_Patterns_Set_Run_* .bat** calls concatenate_files.exe, generate_index_for_run.bat, RapidMiner_ImportFile.bat and RapidMiner_FP_Growth_process.bat
- **9_*_Create_FP_Growth_Patterns_Set_Run_* .bat** is used to combine different combination of runs for creating a combined FP Growth results training sets. Here run combinations used are: run12, run34, run56, run78, run1234, run5678 and run12345678.

Figure 4.25: Parametric view of batch files(3)
5

Results

The prototype was executed a number of times to calculate average execution time and percentage of data reduction at each step. Sub section 5.1 describes the step by step procedure to execute the analysis of trace log file. This will create results in tabular form shown in sub section 5.2. Based on the calculations and evaluations, after observing the results, a conclusion is derived in sub section 5.3 making remarks on the techniques used in implementation of the prototype. To integrate and scale up this solution, sub section 5.3 gives a future scope for the prototype.

5.1 Testing and analysis

Two training sets are created out of GSP and FP Growth output patterns. These need to be used to test the new error trace log file. It is termed as ‘error’ trace log to avoid confusion with the first normal trace log and also because the file is analyzed for containing any error. The error trace log file first undergoes basic pre-processing of removing noise. Noise removal consists of creating $ delimitation file, filtering channels and removing multiline messages retaining the first line of such chunks. The next step is to remove the set up messages which show the startup of the unit since they are not required to be analysed. The aim is to analyze only those messages which are produced out of functional testing of the device. The pre-processed clean error trace log is ready to be tested under the training set. The patterns present in the training set are matched with the messages in the error trace log. If any message pattern is matched, the whole row is marked and replaced by its sequence number and pattern number. Once all the pattern matching is done by GSP training set, these marked rows are removed and saved as another file. The old file is retained for now to understand individual reduction by GSP algorithm. The new file
undergoes pattern matching by FP Growth training set. The output file produced after this step is the final file where both the training sets are matched and all normal sequence patterns (GSP output) as well as frequent item sets (FP Growth output) are removed. This file will have significantly less data as compared to the data which was present in the raw error trace log file. Later this file is given for manual analysis which definitely reduces the effort to find the abnormality in the traces.

5.2 Results

In this section, there are outcome of all functions in detail which will help calculate the percentage for evaluating the performance and success of this thesis. Along with viewing the data reduction at each step, the time required for each process can be noted. The table is divided into two sections: Create Training set and Analysis of trace log file.

i Create Training set

**Evaluation:**
As seen in the table 5.1, the training set creation requires approximately 17 minutes by the automated prototype. Manually, it will take several hours to do so. Few things are needed to be highlighted in the above table. For the technique ‘Generate GSP Patterns for runs’, the corresponding field consists of row with value ‘Run 56’ which has its execution time as 1 hour 27 minutes. This is an exceptional case because of its huge execution time. The reason for such long time being there are many frequent and long patterns present in this run as compared to others. Rest all of the six runs need approximately similar times for their execution. This run may consist of long sequences which makes it produce long candidate sequences to be evaluated. Hence, this particular time is not counted while calculating average for execution time of running GSP algorithm.

Other highlight would be regarding FP Growth algorithm. Unlike GSP, FP Growth is not completely automated. Reason for this is a constraint of the Rapid Miner tool. Rapid Miner 5.3 does not print all frequent itemsets of a run in its output file but lists around 60-70 items with a line appending which says “........and 763 frequent itemsets more”. This makes the output insufficient to be analyzed. This problem will be solved when command line execution for Rapid Miner 6.0 is enabled or when another implementation of the algorithms is used in future versions of our automated analysis tool. Hence, the execution of FP Growth is partially automated. The manual steps include clicking on run button and copying the output to the output file which is created automatically and rename it appropriately. As it can be seen in the table 5.1, the partial automation step together with manual steps takes 30 seconds on an average.
Table 5.1: Calculations: Create Training set

<table>
<thead>
<tr>
<th>Technique</th>
<th>Functions</th>
<th>Process</th>
<th>Average time by automated prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Processing</td>
<td></td>
<td>Check HMI</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dropped messages</td>
<td></td>
</tr>
<tr>
<td>Remove noise</td>
<td></td>
<td>Create $ delimitation file</td>
<td>14 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Filter Channels</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remove multiline messages</td>
<td></td>
</tr>
<tr>
<td>Structure log</td>
<td></td>
<td>Remove dynamic part of messages</td>
<td>1 minute 20 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Index columns</td>
<td>1 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Replace indexes in trace log file</td>
<td>6 minutes 27 seconds</td>
</tr>
<tr>
<td>Creating input</td>
<td></td>
<td>Separate runs</td>
<td>1 second</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transpose</td>
<td>1.7 seconds</td>
</tr>
<tr>
<td>Generate GSP Patterns for runs</td>
<td>1. Concatenate runs</td>
<td>Run 12345678</td>
<td>46.5 seconds</td>
</tr>
<tr>
<td></td>
<td>2. Generate index for run</td>
<td>Run 1234</td>
<td>15 seconds</td>
</tr>
<tr>
<td></td>
<td>3. Import File</td>
<td>Run 5678</td>
<td>50 seconds</td>
</tr>
<tr>
<td></td>
<td>4. GSP process</td>
<td>Run 12</td>
<td>41 seconds</td>
</tr>
<tr>
<td></td>
<td>5. Post process</td>
<td>Run 34</td>
<td>8 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Run 56</td>
<td>1 hour 27 minutes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Run 78</td>
<td>49 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Average</td>
<td>35 seconds</td>
</tr>
<tr>
<td>Create Training set</td>
<td></td>
<td>Concatenate all GSP output patterns</td>
<td>0.7 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remove duplicate patterns</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Convert indexes to message regex</td>
<td>4 minutes</td>
</tr>
<tr>
<td>Generate FP Growth Patterns for runs</td>
<td>1. Concatenate runs</td>
<td>Run 12345678</td>
<td>33 seconds</td>
</tr>
<tr>
<td></td>
<td>2. Generate index for run</td>
<td>Run 1234</td>
<td>27 seconds</td>
</tr>
<tr>
<td></td>
<td>3. Import File</td>
<td>Run 5678</td>
<td>29 seconds</td>
</tr>
<tr>
<td></td>
<td>4. FP Growth process</td>
<td>Run 12</td>
<td>26 seconds</td>
</tr>
<tr>
<td></td>
<td>5. Manual execution</td>
<td>Run 34</td>
<td>35 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Run 56</td>
<td>30 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Run 78</td>
<td>29 seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Average</td>
<td>30 seconds</td>
</tr>
<tr>
<td>Create Training set</td>
<td></td>
<td>Total Average Time</td>
<td>17 minutes 41.3 seconds</td>
</tr>
</tbody>
</table>

75
Sequence patterns produced by GSP were completely useful and matched during pattern recognition in new trace logs. The training has undergone a manual grading by the developers of the HMI department and all the sequence patterns looked important. This proves that GSP is an efficient algorithm to capture useful patterns.

ii Analysis of trace log file

<table>
<thead>
<tr>
<th>Technique</th>
<th>Function</th>
<th>Process</th>
<th>Time taken by automated prototype</th>
<th>Data reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyze Error log AddToHistoryBlocked</td>
<td>Pre-processing</td>
<td>Remove noise</td>
<td>6 seconds</td>
<td>83.01%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remove set up</td>
<td></td>
<td>14.43%</td>
</tr>
<tr>
<td>Error check</td>
<td></td>
<td>Error check by GSP patterns individually</td>
<td>20 seconds</td>
<td>1.45%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Error check after both the patterns combined</td>
<td>9 seconds</td>
<td>0.79%</td>
</tr>
<tr>
<td>Total Time required</td>
<td></td>
<td></td>
<td>35 seconds</td>
<td></td>
</tr>
<tr>
<td>Total normal data removed from trace log</td>
<td></td>
<td></td>
<td>99.32%</td>
<td></td>
</tr>
<tr>
<td>Data removed with only GSP</td>
<td></td>
<td></td>
<td>39.20%</td>
<td></td>
</tr>
<tr>
<td>Data removed by both GSP and FP patterns</td>
<td></td>
<td></td>
<td>73.24%</td>
<td></td>
</tr>
<tr>
<td>Remaining Manual Effort</td>
<td></td>
<td></td>
<td>0.68%</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: Calculations: Analysis of trace log file (2)

<table>
<thead>
<tr>
<th>Technique</th>
<th>Function</th>
<th>Process</th>
<th>Time taken by automated prototype</th>
<th>Data reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyze Error log RouteOptionsNotSet</td>
<td>Pre-processing</td>
<td>Remove noise</td>
<td>6 seconds</td>
<td>84.44%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remove set up</td>
<td></td>
<td>13.81%</td>
</tr>
<tr>
<td>Error check</td>
<td></td>
<td>Error check by GSP patterns individually</td>
<td>20 seconds</td>
<td>0.57%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Error check after both the patterns combined</td>
<td>7 seconds</td>
<td>0.14%</td>
</tr>
<tr>
<td>Total Time required</td>
<td></td>
<td></td>
<td>33 seconds</td>
<td></td>
</tr>
<tr>
<td>Total normal data removed from trace log</td>
<td></td>
<td></td>
<td>99.03%</td>
<td></td>
</tr>
<tr>
<td>Data removed with only GSP</td>
<td></td>
<td></td>
<td>23.90%</td>
<td></td>
</tr>
<tr>
<td>Data removed by both GSP and FP patterns</td>
<td></td>
<td></td>
<td>44.15%</td>
<td></td>
</tr>
<tr>
<td>Remaining Manual Effort</td>
<td></td>
<td></td>
<td>0.97%</td>
<td></td>
</tr>
</tbody>
</table>
5. RESULTS

5.2. Results

Evaluation:
As seen in the tables 5.2 and 5.3, two error trace log files have been analyzed and their individual results are listed. Overall time to analyze and remove normal patterns from the new trace logs is about 33 seconds which is very less as compared to manual removal of normal patterns which takes many hours (or sometimes even days). In the first error trace log, total data reduction is about 99.32% which gives a minimal amount of data (0.68%) to be analyzed manually. In this reduction, GSP patterns have contributed to 39.20% of data reduction and GSP along with FP Growth patterns have made it to 73.24% which is a good amount of reduction. In the second error trace log, 99.03% of data was reduced on the whole which leaves mere 0.97% of manual effort pending. GSP alone has contributed for 23.90% of overall pattern reduction. GSP in conjugation with FP Growth has caused 44.15% of normal data to be eliminated. Finally, the remaining manual effort is calculated as 0.97% which is significantly less. This implies that the implementation works in principle and succeeded in providing minimal effort for manual analysis. The result file with all the normal patterns removed was handed over for manual analysis and the error/abnormality was found in both the trace log files in less than 5 minutes.
Conclusion

Looking at the above calculations, it can be concluded that the work of the thesis achieved success by completing its following tasks:

- Finding an appropriate pattern recognition algorithm
- Analyzing given trace logs by pre-processing and filter as many as normal messages from the abnormal ones
- Implement a working prototype which can assist in error analysis

6.1 Observations

Bar charts, as shown in figure 6.1 and 6.2 show the evaluation of using only GSP, only FP Growth and both the algorithms’ outputs together on the trace log file. It also showcases individual contribution of each algorithm to remove normal patterns from the trace log file to be analyzed.

In both the bar charts 6.1 and 6.2, it can be clearly seen that GSP surely adds an outstanding contribution for normal patterns removal. GSP basically removes sequence of messages rather than individual messages. On the other hand, FP growth removes frequent itemsets which could not be a part of the sequences as they appear individually. Such messages are to be removed but only after the normal sequence patterns are removed. This clearly shows a high priority for using GSP looking at its result and importance in analysis. FP Growth surely performs well which can be seen in the bar charts. However, GSP performs better than FP Growth in both the cases. Nevertheless, we combined results of both the algorithms and mined their patterns together on the trace log file. This
6. Conclusion

6.1. Observations

Figure 6.1: Evaluation of algorithm result for trace log: AddToHistoryBlocked

![Bar chart for AddToHistoryBlocked](image)

Figure 6.2: Evaluation of algorithm result for trace log: RouteOptionsNotSet

![Bar chart for RouteOptionsNotSet](image)

produces an exceptionally good result in both cases. It can be concluded that both the algorithms though individually give good results, perform better when employed together. Since manually induced errors can be detected with rather very minimal manual effort
than earlier techniques, this thesis work has come to its conclusion.

6.2 Future improvements and extensions

The current implementation fulfills the purpose of the thesis of deviating normal patterns from the abnormal ones in the trace log file. But it can be improved further by searching alternate faster approaches and optimizing it. A major improvement in the prototype can be done which concerns usage of Message library. Message library, as mentioned earlier, contains all standard trace messages extracted from the source code. In this implementation, these messages have been converted to regular expressions through automated code. This has been done to compare and search such message patterns in the trace log file ignoring their dynamic part. This procedure highly assists pattern matching in comparison to matching of static messages which do not usually match just due to different parameters. But not all messages have been converted to regex due to variations in their structure. It was somewhat difficult to handle all the cases of the message structure and convert them to regex. Maximum messages have been converted but if more optimizations are done to improve regex, there will be more patterns matched. Also other algorithms could be used that automatically determine the static and dynamic parts of log messages. Secondly, FP Growth algorithm is partially automated due to drawback of the data mining tool. Rapid Miner 5.3 does not include all frequent itemsets in its exported output file. It writes complete output only in the results view of the tool unlike for GSP algorithm which includes all patterns in its exported output file. This can be optimized when Rapid Miner is updated to be executed from command line or by using another implementation (maybe a custom one).

The prototype implemented has a good scope for its usage in the department of HMI testing. As soon as any trace log file is produced after regular HMI tests, this prototype can be triggered to start processing and analyzing the new trace log file. Training set can be reused unless there are major changes in the functional behavior or messages in the source code. If there are any changes, the training set can be recreated immediately which requires an input of normal trace log file with maximum 8-9 runs which is good enough to derive normal patterns. We can also create a greater number of logs and different use cases for the training phase of the process. The error analysis is partially automated since manual analysis is important. More efforts can be taken to completely automate the analysis once the testing is stable.


Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende **Masterarbeit** selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.


Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: **13.01.2016**

Unterschrift: ..........................................................

---

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people’s work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.